RY
ATE SCHOOL
RNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

K7827

THE DESIGN AND IMPLEMENTATION OF A SPECIFICATION LANGUAGE TYPE CHECKER

by

Robert George Kopas

June 1989

Thesis Advisor: Valdis Berzins

Approved for public release; distribution is unlimited

T244018

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report: Approved for public release; distribution is unlimited.
2b Declassification/Downgrading Schedule	
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization: Naval Postgraduate School; 6b Office Symbol (If Applicable): 52	7a Name of Monitoring Organization: Naval Postgraduate School
6c Address (city, state, and ZIP code): Monterey, CA 93943-5000	7b Address (city, state, and ZIP code): Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number
8c Address (city, state, and ZIP code)	10 Source of Funding Numbers: Program Element Number; Project No; Task No; Work Unit Accession No

11 Title (*Include Security Classification*)

The Design and Implementation of a Specification Language Type Checker

12 Personal Author(s)

Kopas, Robert G.

13a Type of Report	13b Time Covered	14 Date of Report (year, month,day)	15 Page Count
Master's Thesis	From To	June 1989	219

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes		
Field	Group	Subgroup

18 Subject Terms (*continue on reverse if necessary and identify by block number*)

Specification Language, Type Checker, Attribute Grammar, Software Engineering, SPEC Specification Language.

19 Abstract (*continue on reverse if necessary and identify by block number*)

The purpose of this thesis is to design a type checker for the SPEC language and to investigate its implementation using an attribute grammar tool. SPEC is a formal language for writing black-box specifications for large software systems. The type checker is a software tool which verifies the semantic accuracy of the declarations and their uses in a SPEC source program. The design specifically addresses language features which are especially important for large software system specification such as generic parameters, name and operator overloading, subtypes, importation and inheritance. Additional discussion is provided concerning the handling of the "non-block structured" nature of the specification language. This thesis implements two of the three aspects of type checking--name analysis and error reporting. Additionally, a definitive framework is laid for the final aspect, type consistency analysis.

20 Distribution/Availability of Abstract	21 Abstract Security Classification	
[X] unclassified/unlimited [ ] same as report [ ] DTIC users	Unclassified	
22a Name of Responsible Individual: Prof. Valdis Berzins	22b Telephone (*Include Area code*): (408) 646-2461	22c Office Symbol: Code 52Bz

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted — All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

The Design and Implementation of a Specification Language Type Checker

by

Robert George Kopas
Lieutenant, United States Navy
B.S., Purdue University, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
JUNE 1989

ABSTRACT

The purpose of this thesis is to design a type checker for the SPEC language and to investigate its implementation using an attribute grammar tool. SPEC is a formal language for writing black-box specifications for large software systems. The type checker is a software tool which verifies the semantic accuracy of the declarations and their uses in a SPEC source program. The design specifically addresses language features which are especially important for large software system specification such as generic parameters, name and operator overloading, subtypes, importation and inheritance. Additional discussion is provided concerning the handling of the "non-block structured" nature of the specification language. This thesis implements two of the three aspects of type checking--name analysis and error reporting. Additionally, a definitive framework is laid for the final aspect, type consistency analysis.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. OBJECTIVES 1
B. RESEARCH QUESTIONS 2
C. THE HISTORY AND SUCCESSFUL APPLICATIONS OF ATTRIBUTE GRAMMARS TO SIMILAR PROBLEMS 2
1. The Context Free Nature of SPEC--A Pretty Printer 3
2. MSG.84 and MSG.85--A Translator for a Specification Language 4
3. The Design of a Compiler--Farrow 1984 4
4. Syn--A Graphical Representation of Bachus Naur Form 5
5. The Chameleon Architecture 5
D. A SPECIFICATION TYPE CHECKER AND "CASE" 6
1. Description of CASE 6
2. Benefits for CASE 6

II. BACKGROUND 7
A. TYPE CHECKER 7
1. Definition 7
2. Scope Considerations 8
B. ATTRIBUTE GRAMMARS 8
1. Attribute Grammars 8
a. Definition 9
b. Synthesized and Inherited Attributes 9
c. Semantic Functions 9
2. Automatic Parsing of Attribute Grammars--Application Generators 9
a. Definition and Overview 10
b. The Semantic Tree 11
c. Semantic Analysis 11
d. Advantages and Disadvantages 11
e. Existing Application Generators 13
C. THE KODIYAK APPLICATION GENERATOR 13
1. Justification for use 13
2. A General Description of the Kodiyak Language 14
a. Format 14
b. Comments 15
c. Lexical Scanner Section 15
d. Attribute Declaration Section 18
e. The Attribute Grammar and Semantic Function Section 20

DUDLEY KNOX LIBRARY
NAVAL PO... SCH
MONTEREY, CALIFORNIA 93

f. Using the Kodiyak Translator. 24
D. THE SPEC LANGUAGE. 25
1. The Event Model 25
2. The SPEC Language 29
a. Functions 29
b. State Machines 30
c. Types 31

III. SYSTEM DESIGN. 35
A. SPEC LANGUAGE TYPE CONSISTENCY CONSTRAINTS. 35
1. SPEC Language Semantic Issues. 35
a. Definitions. 35
b. Scoping. 36
c. Naming Constraints. 37
d. Type Consistency Constraints. 38
B. CONCEPTUAL MODEL. 39
1. Requirements. 39
2. Model. 41
C. DESIGN CONVENTIONS. 46
1. Attribute Naming. 46
2. M4 Macro Abstractions. 47
a. Attrib_psg.m4 48
b. Mymac.m4 51
(1) Declaration Group. 51
(2) Symbol and Visibility Tables Group. 51
(3) Attribute Evaluation Group. 51
c. Myconst.m4 51

IV. IMPLEMENTATION. 52
A. SEMANTIC INFORMATION STORAGE STRUCTURES. 52
1. Module Types. 52
2. Symbol Table. 53
a. Textual Names 54
b. Parameters. 54
c. Results. 55
d. Classes. 55
3. Visibility Tables. 56
a. Visible Types 56
b. Visible Names. 56
B. MAJOR ATTRIBUTES. 57
1. Error Reporting. 57

a. verror_message. 57
b. Declaration Errors. 57
c. Error Concatenation. 58
2. Building the Symbol Table. 58
3. Extended Types. 59
C. NAME ANALYSIS. 59
1. Checking if an identical declaration exists. 60
2. Making a new declaration. 60
3. Reporting an error. 60
D. IDENTIFYING ERRORS TO THE USER. 61

V. EXTENSIONS. 62
A. TYPE CONSISTENCY ANALYSIS. 62
1. Seeking the Correct Symbol Table Entry. 62
2. Obtaining a Symbol Table entry's type. 63
3. Determining if an Operator is defined. 63
4. Reporting Errors. 63
B. SPECIAL SPEC LANGUAGE ISSUES. 64
1. Inheritance & Instance Declarations. 64
a. Preprocessor Usage. 65
b. Error Reporting Drawback. 65
c. Potential Advantages. 65
2. Importation & Exportation. 66
C. IMPROVED ERROR REPORTING. 67
D. SUBTYPES. 67
E. VARIABLE ARGUMENT OR PARAMETER LISTS. 68

VI. CONCLUSIONS. 69
A. INTEGRATION INTO A PROGRAMMING ENVIRONMENT. 69
B. EVALUATION OF THE TYPE CHECKER. 70
1. Kodiyak Deficiencies. 70
2. Kodiyak Benefits. 71
C. FUTURE WORK. 72
1. Extensions of the current implementation. 72
2. Incremental Type Checking. 72
D. GUIDELINES FOR EXTENDING KODIYAK. 73

APPENDIX A - SPEC GRAMMAR 74

APPENDIX B - CODE. 89
1. MAKEFILE 89

2. ATTRIB_PSG.M4 89
3. MYMAC.M4 95
4. MYCONST.M4 97
5. HEAD.M4 98
6. TAIL.M4 98
7. MYLIB.C 98
8. SPEC.M4 109

APPENDIX C - SYNTACTIC ERROR PRODUCTIONS. 174

APPENDIX D - TYPE CHECKER ATTRIBUTES 200

REFERENCES 203

BIBLIOGRAPHY 205

INITIAL DISTRIBUTION LIST 207

LIST OF FIGURES

2.1 Sample Lexical Definitions 16

2.2 Lexical Section Operators 17

2.3 Regular Expressions with Definition Expansion 18

2.4 Sample Attribute Declarations 19

2.5 Rule Template 20

2.6 Attribute Naming and Rule Concepts 21

2.7 Kodiyak Operators 22

2.8 Kodiyak "If..Then..Else" Construct 23

2.9 Kodiyak Functions 24

2.10 Function Example 30

2.11 Machine Example 31

2.12 Immutable Abstract Data Type 32

2.13 Mutable Abstract Data Type 34

3.1 Conflicting Name Bindings 40

3.2 Limiting Type Visibility 42

3.3 Localizing a Map 44

3.4 Name Layering 44

3.5 Result Values Transitive Dependency 45

3.6 Passing Up an Attribute 49

3.7 Passing Down an Attribute to "n" Non-terminals 49

3.8 Passing Over an Attribute to "n" Non-terminals 50

3.9 Passing an Attribute in order to "n" Non-terminals 50

3.10 Weaving an Attribute to "n" Non-terminals 50

4.1 Example of a Pattern 54

ACKNOWLEDGMENT

This thesis is dedicated to my loving wife, Sue. Thank you for all your patience and understanding during the long hours devoted to its design, implementation and writing. Most importantly, however, thank you for just being there and being you.

I. INTRODUCTION.

The field of Software Engineering is a growing area of interest in computer science. Many systems are currently being developed, using various methodologies, to support large scale software development. One of these systems uses the functional specification language SPEC to define the syntax and semantics of these projects. SPEC is a language, developed by Berzins [Ref. 1], for writing black-box specifications for the components of the software system in the functional specification stage of software development. To increase the reliability of this code a type checking tool has been developed to semantically validate the various type declarations used in the specification. This thesis describes the methodology used and the actual implementation of this type checker.

A. OBJECTIVES.

The primary objective of this thesis is to design and implement a language dependent type checker for the specification language SPEC. Appendix A and [Ref. 1] contain a listing of the grammar for the SPEC language. The code for the type checker is written entirely as an attribute grammar. This code is then compiled using the Kodiyak Application Generator, producing executable code.

It is also desired that the type checker be easily modifiable in the event that extensions are incorporated into the SPEC language. By using an attribute grammar to specify the program and an application generator to generate the code, the program can

be modified with relative ease. The efficiency of the code is not a limiting factor, because optimizations may be coded at a later date.

The final objective is to abstract the process used to develop the type checker and develop an algorithm for producing type checkers for other specification languages. This algorithm will be a direct result of the insight gained from the design and implementation of the SPEC language tool.

B. RESEARCH QUESTIONS.

There are two pertinent research questions for this thesis. First, and foremost, what are the underlying issues in the design and implementation of a type checker for a specification language using an application generator? What is the impact of these issues on portability, readability, modifiability and efficiency of the code?

Finally, how would this type checker be best integrated into a programming environment? What facilities must this programming environment provide? Should the type checker be optimized to provide better execution speed?

C. THE HISTORY AND SUCCESSFUL APPLICATIONS OF ATTRIBUTE GRAMMARS TO SIMILAR PROBLEMS.

Attribute grammars are frequently used for specifying the semantic meaning of source programming languages. Additionally, with the aid of application generators, these grammars have been used to generate compilers and other tools for the recognition and transformation of programs coded in these source languages.

1. The Context Free Nature of SPEC--A Pretty Printer.

All language grammars are classified by the complexity of the productions that produce that language. The Chomsky Heirarchy, named after Noam Chomsky, divides languages into four classes--type 0 (unrestricted) grammars, type 1 (context sensitive) grammars, type 2 (context free) grammars and type 3 (regular) grammars. Type 0 grammars contain the set of all languages, type 1 contains a subset of that, etc. Of these four classes, type 3 (context free) and type 4 (regular) are of the most interest in programming language design because they can be used to describe the structure and basic symbols of a program. Most standard computer languages in use today can be described with a context free grammar since efficient parsing algorithms are known for this class of languages. The SPEC specification language is described using a context free grammar.

A thesis recently completed at the Naval Postgraduate School [Ref. 2] developed a program using the Kodiyak application generator and an attribute grammar to print SPEC specifications in a properly formatted / indented manner. The Kodiyak application generator is a tool for converting context free attribute grammars into an executable program. This "Pretty Printer" program demonstrated that SPEC is indeed context free because the Kodiyak tool could convert the attribute grammar representation into executable code which formats SPEC specifications. The thesis also demonstrated that attribute grammars are a feasible way to create tools for the SPEC language.

2. **MSG.84 and MSG.85--A Translator for a Specification Language.**

MSG.84 and MSG.85 are specification languages developed at the University of Minnesota. They have been used extensively in software engineering classes for specifying software systems. In the Spring of 1984, a translator was written to translate these specifications into the Lisp-like DBL [Ref. 3] assertions. This translator, from which the Kodiyak application generator evolved, revealed some design flaws in the MSG language. These flaws were corrected and a translator now exists for the conversions. Additionally, a reverse translator was written to convert the DBL assertions produced into MSG. This process of translation and reverse translation insured that the translation to DBL was "lossless" and that the semantic meaning of all the MSG constructs was preserved.

3. **The Design of a Compiler--Farrow 1984.**

Attribute Grammars have long been championed as a promising basis for compiler writing systems. Many different compiler-compilers such as Yacc [Ref. 4], Linguist [Ref. 5] and Kodiyak [Ref. 6] have been developed. One of the major drawbacks however, has been the inability of these compilers to compete in the commercial market with those compilers produced by other means due to their memory requirements and speed. The Pascal-86 compiler developed by Intel Corporation is based on an attribute grammar and application generator. It was successfully marketed as a production compiler and was developed in a two stage process. The first phase was developed using the Linguist-86 application generator and performed semantic analysis, storage allocation and translation to intermediate code. The second part of the compiler

takes the intermediate code produced by the semanticist unit and generated 8086 microprocessor object code. The compiler has been marketed succesfully and it was noted that the development process was significantly enhanced due to the use of an attribute grammar.

4. **Syn--A Graphical Representation of Bachus Naur Form.**

Syn is a translator developed with the Kodiyak application generator that translates a grammar expressed in the Bachus Naur Form into directives in the PIC graphics language. The translator required approximately two man-weeks of work to implement by a user initially unfamiliar with the Kodiyak application generator [Ref. 6]. The significant time savings realized by the use of an application generator in the development of this tool reaffirms that application generators are an effective tool for developing large applications.

5. **The Chameleon Architecture.**

The Chameleon project [Ref. 7] is an ongoing project at Ohio State University that is developing an architecture to support the specification, construction and use of data translation tools. The architecture's primary tool is an application generator that will take attribute grammar specifications and produce the tools needed to translate the data. An application generator was choosen as the primary tool in this architecture due to the readability and ease of understanding of the attribute grammar specifications.

D. A SPECIFICATION TYPE CHECKER AND "CASE".

1. Description of CASE.

CASE is an acronym for Computer Aided Software Engineering, an area of ongoing research at the Naval Postgraduate School. Some of the projects currently being developed in this area include a Software Rapid Prototyping System, Syntax-directed editors for SPEC and formatters for the SPEC language. Eventually, all of these tools will be integrated into an environment to aid the software engineering process and specifically ADA program development. It is also anticipated that this type checker and its principles will be integrated with a syntax directed editor to build a superior editing, type checking tool.

2. Benefits for CASE.

A type checker would be an extremely valuable addition to any CASE environment. Since it has been proven that many design errors manifest themselves as a type inconsistency, the type checker would assist in the identification of errors that could defeat the reliability of the specification.

II. BACKGROUND.

A. TYPE CHECKER.

1. Definition.

A Type Checker is a tool used to validate the semantic accuracy of the uses and declarations of name structures in a program. There are two types of type checkers currently in use today. The kind of type checker used for a language is often dependent on the features of that language. The first kind, a dynamic type checker, executes concurrently with the program and checks the validity of dynamically declared variables as they are encountered. Since this tool runs as part of the executable program, the program performance is degraded and errors are reported during run time instead of before the compile - link cycle.

The second kind of type checker is a static type checker. This tool takes as input a file or multiple files containing the source program / specification and provides the user with a report detailing any type inconsistencies that were discovered. It can be run before a program is compiled and reports inconsistencies so that they may be corrected before the compile - link cycle.

The process of validating the semantic accuracy of the structures in a program is a multi-part process. The first part, called name analysis, entails finding the definition of that name applicable to each use of the name. If there has been no definition of that name, the type checker will either make a definition based on the name's use or report an

error. The other two phases deal with obtaining the operator being used and confirming that the results of the name analysis are indeed allowable with the given operator. Additionally, a type checker must consider various language features such as operator overloading, name scoping and the binding method used.

2. Scope Considerations.

The scope of a name is the portion of the program over which it may be used [Ref. 8]. Many languages, called block structured languages, allow the nesting of various names within themselves. The most recent occurence defining the currently available definition of that name. Another constraint imposed by scope is whether or not a variable is visible inside of the structures that are declared inside of it. Both of these constraints and others must be considered in the design of the type checker.

B. ATTRIBUTE GRAMMARS.

1. Attribute Grammars.

Attribute grammars were introduced by Knuth [Ref. 9] and advocated as a means of translating grammar specifications into executable code. They have been used repeatedly to develop compilers, compiler-compilers (application generators) and other useful tools. One of the most significant features of attribute grammars is their readability. Attribute grammars are very similar to the Bachus-Naur form (BNF) of representing the syntactic structure of a program and tends to be self-documenting since they represent a relation between the syntax and semantics of the language.

a. Definition.

An attribute grammar is based upon a context-free grammar G = (N, T, P, Z) and associates a set A(X) of attributes with each symbol, X, in the grammar G. The context-free grammar is used to represent the syntactic structure of the language while the attributes are used to represent context-sensitive (semantic) properties of the language.

b. Synthesized and Inherited Attributes.

The attributes of an attribute grammar may be divided into two classes. The first of these, synthesized attributes, are those attributes of symbols on the Left hand side of a production that are derived from the return value of the elements on the right-hand side of the production. Conversely, inherited attributes are those attributes of symbols on the right hand side whose values are derived from the values of the attributes of the symbols on the left hand side.

c. Semantic Functions.

Each rule in an attribute grammar has semantic functions associated with it that define the values of some attributes in the production in terms of other attributes in the function. These functions resolve the actions that the application generator takes upon recognizing the production associated with them and define the values of all inherited and synthesized attributes.

2. **Automatic Parsing of Attribute Grammars--Application Generators.**

Application generators have many different uses. They have been used frequently for the implementation of compilers, the verification of the context sensitive constraints

on languages and are becoming popular for the design and implementation of various other tools. One of their most significant advantages is that they let the user customize and reuse a general software design easily.

a. Definition and Overview.

Application generators are tools that produce executable code from a grammar specification. The executable program produced will model the semantic meaning of the grammar specification precisely. The application generator consists of two parts, a variant part and an invariant part. The invariant part consists of fixed assumptions about the domain or implementation such as the source language. The variant part of the application generator corresponds to the attribute grammar specification of the system that is to be produced.

The process of using the application generator begins with the attribute grammar specification. This specification is generally simpler, in both syntax and semantics than the programming language it specifies. Acting much like a language compiler, the application generator takes this specification and produces code in some invariant language (e.g., "C") which is then compiled with a standard compiler to produce the executable application. An end-user may then take this executable application and provide it with input from which the application will produce whatever the original specification specified.

There are many different considerations in the choice of an application generator for a specific task. The first and foremost of these is if the application generator can perform the task desired. Other considereations include what built-in types

and operators are available in the applications generator, the readability of the specification language (attribute grammar) used by the tool, the availability of the generator itself and the attribute evaluation method.

b. The Semantic Tree.

The basic principle of operation behind an application generator and its associated attribute grammar is that any program can be represented by a semantic tree. This tree will contain nodes, the interior ones representing productions and the leaves representing terminal symbols in the target language. Associated with each node is it's attributes.

c. Semantic Analysis.

The process of evaluating each node in the semantic tree is called semantic analysis. As the attributes of each node are evaluated, the semantic functions are executed and any side-effects associated with them are performed. Different algorithms have been derived for resolving the attributes in the semantic tree and most of them have been implemented in at least one application generator. The choice of the algorithm depends on the properties of the attribute grammar and the qualities desired in the resultant product [Ref. 6]. To ensure that the translator produced by the application generator performs exactly as desired, it is imperative that the method used to evaluate the attributes be understood.

d. Advantages and Disadvantages.

Application generators produce tools that are more reliable than a conventially coded tool because of their very nature. Since it accepts a specification of the tool to be

produced and uses well-known techniques, it is less likely to accept syntactically incorrect input and generate unexpected output or terminate abnormally [Ref. 6]. Since there is a close link between the syntax and semantics of the specification, the volume of code required is reduced and it is more repairable. The application generator tends to be self-documenting because its source code allows users to quickly determine the syntactic requirements of programs. If an application generator is constructed properly it can be very portable. Typically it can be written in its own language and it produces an appropriate, portable target language, the application generator can be transfered to other computers with relative ease.

The most important advantage of an application generator is its ability to cut down on the time and cost to build a tool with it. Since a programmer's productivity is largely constant in the number of lines of code produced per unit time [Ref. 10], a tool that generates a program equivalent to a high level language program with less actual lines of code will increase productivity. Additionally, since there is generally a close correlation between the Bachus-Naur representation of the grammar and the specification input to the application generator, the time involved in the development of the program is decreased as it is with most fourth generation languages [Ref. 11].

A major drawback of application generators is that since they are table driven, they tend to produce executable code that is not as efficient as equivalent "hard-coded" tools. Additionally, since the value of the attributes used must be copied between attributes so that these values may reach the root to be output, bulk is added to the program's specification. Both of these problems are currently being researched. Some

solutions have been put forth. To reduce the number of copy rules (and thus increase readability), a macro preprocessor may be used. Additionally, research has proven that hard-coding the tables used during the evaluation process tends to produce a speedup factor of 6-10% [Ref. 12].

e. Existing Application Generators.

Many different application generators have been developed. Some of the most popular include the HLP (Helsinki Language Processor) system [Ref. 13], Delta [Ref. 14], Mug2 [Ref. 15], Aparse [Ref. 16] Gag [Ref. 17], YACC [Ref. 4], Linguist-86 [Ref. 5] and Kodiyak [Ref. 6]. Each uses a different algorithm for evaluating the attributes of the semantic tree and accepts different classes of attribute grammars.

C. THE KODIYAK APPLICATION GENERATOR.

The Kodiyak application generator was developed at the University of Minnesota and is intended for building prototype languages and translators. The specification describes all aspects of translation: input scanning, parsing, semantic processing and output. It has been used to build translators for other specification languages, text processing tools, database query languages and a pretty printer for the SPEC language.

1. Justification for use.

Kodiyak is an exceptional language translator that integrates the functions of the YACC [Ref. 4] parser generator and LEX [Ref. 18] scanner generator into a comprehensive whole, hiding the procedural and interface details of these tools. Its compact attribute grammar specification describes every aspect of the translation process, produces portable "C" language code and then compiles that into executable code.

Additionally, Kodiyak can process it's input through a macro-preprocessor allowing repetitive code to be replaced by a single statement thus improving a programs' readability. Kodiyak allows evaluation of the largest class of attribute grammars and contains built-in types capable of specifying symbol tables. Its many features, capabilities, portability, and availability make it an ideal tool for the implementation of this thesis.

2. A General Description of the Kodiyak Language.

All of the points covered in the following section come directly from Appendix A of Herndon [Ref. 6] Kodiyak Program Layout. It is intended to describe the operation of the Kodiyak translator in sufficient detail to facilitate understanding of the type checker code. It is not intended to be a detailed reference. If further or more detailed information is needed it is recommended that Herndons' doctoral dissertation [Ref. 6] be consulted.

a. Format.

Every Kodiyak program has three sections. The first section describes the features of the lexical scanner that is to be used to translate the source text into tokens and operator precedences and associativities for those tokens. The second section declares the names and types of the attributes associated with each grammar symbol. The third section contains the grammar and attribute equations that define the translation. These sections are separated by a double percent symbol ("%%") on a line by itself.

b. Comments.

There are two forms of comments in Kodiyak. The first is the C and PL/I style comment delimined by "/*" and "*/". The second is introduced by an exclamation point "!" and continues to the end of the line.

c. Lexical Scanner Section.

Each statement in the lexical section of a Kodiyak program describes the terminal symbols of the translation in some way. The primary function of statements in this section is to specify the terminal symbols of the grammar, and how input text is to be transformed into these symbols. The secondary function of this section is to specify a set of operator precedences to be used with the grammar.

The transformation of input text to terminal symbols is denoted by a regular expression. Figure 2.1 shows examples of lexical definitions. These definitions are an excellent sampling of the typical definitions. The first definition demonstrates the general format of a lexical definition. The second definition demonstrates how a specific string can be recognized and the third definition shows how a changeable string of text, such as a variable name, may be recognized. The rules are examined in the way they are listed, thus implying precedence. The first rule that is recognized will determine the terminal symbol (token) that will be returned.

```
TERMINAL_NAME      : REGULAR_EXPRESSION

        ! Format for a lexical definition.
        ! A Terminal_name is specified by the Regular_expression.


BEGIN                   : "BEGIN" | "begin"

        ! The terminal symbol BEGIN is recognized if either "BEGIN" or
        !    "begin" is scanned from the input.


ID                      :[A-Za-z][A-Za-z0-9]*

        ! The terminal symbol ID is recognized if a string starting with
        !    an alphabetic character followed by zero or more alphanumeric
        !    characters is scanned from the input.
```

Figure 2.1
Sample Lexical Definitions.

Operator characters are an extremely important part of any regular expression. They allow ways for specifying choices, repeating characters and ranges of characters. All valid operator characters used in Kodiyak regular expressions are enumerated in Figure 2.2. If you desire additional information on construction of regular expressions and further examples, the original Lex documentation [Ref. 18] provides an authoritive source.

Operator Symbol	Meaning
:	Delimiter between Token name and regular expression.
\	By preceding an operator character with the backslash, the operator will be recognized as a text character.
"	Whatever is contained between a pair of quotes is text characters.
[ ]	Indicates a character class. Any character between the brackets will be recognized. The only operator symbols having meaning between brackets are "-", "\" and "^".
^	If the Caret Symbol appears outside of a set of brackets, the string following it must appear at the beginning of a line to be matched. If it appears as the first character after a left bracket, it indicates that the resulting string is to be complemented. (i.e. [^abc] matches everything except a,b or c.
+	Symbolizes an expression that is to be repeated one or more times.
*	Symbolizes an expression that is to be repeated zero or more times.
|	Indicates alternation. It may be interpreted verbally as an "or".
()	Parenthesis are used for grouping.
$	If the very last character of a regular expression is the dollar sign, the expression will only be matched at the end of a line.
/	The forward slash between two regular expressions means that the terminal symbol is only matched if the first regular expression is immediately followed by the second regular expression.
?	The question mark precedes an optional part of an expression.
-	The dash operator specifies ranges.
.	The period operator matches any character.
{}	Curly Braces specify either repetition or definition expansion.

Figure 2.2
Lexical Section Operators.

Kodiyak also provides ways to increase the readability of regular expressions. By defining the basic lexical classes (digits, alphabetics, integers, etc.), the regular expressions may be made more readable. Figure 2.3 provides an example of this feature.

```
%define Letter          : [a-zA-Z]
%define Int             : {Digit}+
%define Alphanum        : [{Digit}{Letter}]

COMMENT                 : "--".*"\n"
NAME                    : {Letter}{Alphanum}*
```

Figure 2.3
Regular Expressions with Definition Expansion.

d. Attribute Declaration Section.

The attribute declaration section of a Kodiyak program declares the attributes used in the program and their types. In this version of Kodiyak, no other statements may be present in this section, though it is expected that declarations of constant functions and external functions and procedures will eventually be allowed in this section.

Kodiyak supports two primitive data types for attributes: strings and integers. Strings may have arbitrary length and may be concatenated to form longer strings. All simple mathematical functions (i.e., addition, subtraction, multiplication and division) may be applied to integers.

Kodiyak also supports higher order types. These types are called maps, and define functions that may map any primitive type to any other type. Maps are extremely flexible and important to the type checker. They can be mapped onto other maps to form

something similar to high level languages record structures. Figure 2.4 shows a some sample attribute declarations.

```
ID      {
                    type            : string;
                    %text           : string;
                    %line           : int;
                    value           : int;
        }


EXPR    {
                    type            : string;
                    e_valid         : string -> int;
        }
```

Figure 2.4
Sample Attribute Declarations.

This figure declares that four attributes (type, %text, %line and value) are to be associated with the grammar symbol ID and that two attributes (type and e_valid) are to be associated with expressions (EXPR). Furthermore, attributes type and %text are attributes that may take on string values; %line and value may take on integer values and attribute e_valid is a map from a string ("true" or "false") to an integer value (1 or 0).

Figure 2.4 also demonstrates two very important concepts in Kodiyak. First, since an identifier (ID) is normally a terminal symbol and an expression (EXPR) is a non-terminal, both terminals and non-terminals can have attributes. Secondly, terminal symbols can have two special, predefined attributes associated with them, "%text" and "%line". These attributes are initialized when the terminal symbol is recognized to be the actual text scanned and the input line on which the text was found.

e. The Attribute Grammar and Semantic Function Section.

The final section of a Kodiyak program defines the syntax and semantics of the translation. It consists of a set of rules and sets of equations defining evaluation rules for the attributes. There is one distinct start symbol for the rules and it is defined as the symbol on the left hand side of the first rule in the grammar. This symbol is unique and it may not appear on the right hand side of any rule in the grammar.

Rules in Kodiyak are defined in a form that is very similar to Bachus-Naur Form (BNF). Figure 2.5 defines a rule which specifies that the non-terminal symbol "non" will be recognized if the symbols "sym1", "sym2" and "sym3" appear in sequence. Additionally, if the symbol "non" is recognized, the semantic functions defined between the curly braces will be computed during the attribute evaluation process.

```
non :    sym1 sym2 sym3
                        {
                         ! semantic functions go here.
                        }
```

Figure 2.5
Rule Template.

The semantic functions must have a way of specifying exactly what attributes are to be used during the determination process. In Figure 2.6 one rule has been used to demonstrate the three different ways of accessing the same attributes. In the first part of the rule, a production allowing an expression to be recognized when two expressions are separated by a plus ("+") sign is detailed. Associated with it is a semantic function stating the value attribute of the expression on the left hand side of the rule (specified by

the "$$.value" notation) will be assigned the contents of the first expression's value attribute ("$1.value") added to the contents of the second expression's value attribute ("$3.value"). The second and third notation show the exact same effect using subscripting. In these examples, the subscript refers to each occurence of the non-terminal. If a subscript is left out, as in the second notation, the non-terminal is assumed to refer to the non-terminal on the left hand side of the equation.

```
expr    :       expr '+' expr
                {
                 $$.value = $1.value + $3.value
                }

        |       expr '-' expr
                {
                 expr.value = expr[2].value - expr[3].value
                }

        |       expr '*' expr   %prec multiply
                {
                 expr[1].value = expr[2].value * expr[3].value
                }
```

Figure 2.6
Attribute Naming and Rule Concepts.

Kodiyak has a rich set of operators. Besides the various arithmetic and string operators detailed previously, it also provides nine logical operators. Figure 2.7 enumerates all of the operators that are presently available.

Operator Symbol	Meaning.
*	Multiplication
-	Subtraction
/	Division
^	Concatenation
[ ]	Concatenation
<	Less than
>	Greater than
==	Equal to
<>	Not equal to
<=	Less than or equal to
>=	Greater than or equal to
&&	Logical and
||	Logical Or
~	Logical negation

Figure 2.7
Kodiyak Operators.

Kodiyak also provides a means of specifying a statement equivalent to the "IF" construct used in high level languages. This construct has different syntax than most languages, but is logically equivalent. Figure 2.8 demonstrates this construct using the "expr" rule introduced in Figure 2.6.

```
expr    :           expr '/' expr
                    {
                     $$.value = ($3.value <> 0
                                 -> $1.value / $3.value
                                 #  s2i("0")
                    }
```

Figure 2.8
Kodiyak "If..Then..Else" Construct.

The above example assigns to the value attribute of the left part symbol the contents of the first expression divided by the contents of the second expression if the contents of the second expression are not equal to zero. If the second expression's contents are equal to zero, a value of 0 is assigned to the resultant expression's "value" attribute. The "else" ("#") clause also introduces another very important feature of Kodiyak--built in functions. In Figure 2.8, the string to integer ("s2i") function was called to convert a string value into an integer. Kodiyak's standard functions are enumerated in Figure 2.9.

Function Name	Purpose
fmt(format,arg1,...)	Generates a string in the format defined by the "format" parameter, with arg1,... substituted.
i2s(integer)	Converts an integer value to a string representation.
s2i(string)	Converts a character string to an integer.
len(string)	Returns the length of a string.
inputfile(0)	Returns a string naming the input file.
outputfile(0)	Returns a string naming the output file.
basename(file:string)	Returns a copy of a string without dotted extension.
%output(val:string)	Val is written to the standard output
%error(val:string)	Val is written to the standard error.
%assert(condition:boolean, message : string)	If the value of condition is false, Kodiyak prints message to the standard error file and terminates, otherwise, the procedure does nothing.
%outfile(name : string, val : string)	Val is written onto the file "name". If name is null, val goes to stdout.
%errfile(name : string, val : string)	Val is written into the file "name". If name is Null, val goes to stderr.

Figure 2.9
Kodiyak Functions.

f. Using the Kodiyak Translator.

The Kodiyak compiler creates and processes many files. Among them are files that are processed by YACC, LEX, and by the C compiler. The Kodiyak compilation process also depends upon two predefined files. The first is the Kodiyak library. This contains functions for creating the parse tree, evaluating attributes,

concatenating strings, etc. The second file is the user library. This is a set of C functions that the programmer may define for himself.

The command to invoke the Kodiyak translator is "k program.k" where "program.k" is the kodiyak program to be compiled. Files to be compiled should have the extension ".k" or the compiler may not accept it. Kodiyak programs may also have the extension ".m4" if the program is to be run through the macro-preprocessor prior to Kodiyak compilation.

After the program is compiled, (assuming no errors are present), the resulting object code will be in the file "program" in the current directory.

D. THE SPEC LANGUAGE.

SPEC is a formal language for writing black-box specifications for components of software systems. SPEC uses the event model to define the black-box behavior of proposed and external systems. Black-box specifications are developed for the external interfaces of the system in the functional specification stage of software development, and for the internal interfaces in the architectural design stage. Discussion of the event model and the SPEC language, extracted from [Ref. 1:pp. 3.1-3.15], follows. Appendix A and [Ref. 1] contain a listing of the grammar for the SPEC language.

1. The Event Model.

In the event model, computations are described in terms of events, modules and messages. An event occurs when a message is received by a module at a particular instant of time. A module is a black box that interacts with other modules only by

sending and receiving messages. A message is a data packet that is sent from one module to another module.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. A module has no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the kinds of events that can occur at the module along with its response to each kind of event. Each kind of event corresponds to a different of incoming message. Each response consists of the later events directly triggered by a given initial event.

Any module accepts messages one at a time, in a well-defined order that can be observed as a computation proceeds. Message transmission is assumed to be reliable. Messages can have arbitrarily long and unpredictable transmission delays. The order of messages arriving is normally not under control of the designer.

In the event model each module has its own local clock. The local clocks of different modules are not necessarily synchronized with each other. Each event occurs at a well-defined instant of time, which is the time at which the destination module receives a message, according to its own local clock. The length of time between two events is precisely defined if both events occur at the same place. The length of time between two events at different locations can be estimated in terms of two readings of the same clock, but this is only an approximation because of unpredictable message delays in obtaining remote clock readings. The only kind of time interval meaningful in the event model is

the duration between two events. There is no way to distinguish between computation delay and communication delay in the event model.

Each message has a sequence of zero or more data values associated with it. The other attributes of a message are its name, its condition and its origin. All of these attributes are single valued. Exceptions are modeled as messages by means of a condition attribute, which can take on the values "normal" and "exception". The condition of a message expressing a normal request for service is "normal". The condition of a message reporting an abnormal event somewhere is "exception", in which case the name of the message is the name of an exception condition.

The response of a module to a message is completely determined by the sequence of messages received by the module since it was created. A module is mutable if the response of the module to at least one message it accepts can depend on messages that arrived before the most recent incoming message. A module is immutable if the response of the module to every possible message is completely determined by the most recent incoming message. Mutable modules behave as if they had internal states or memory, while immutable modules behave like mathematical functions. A module is immutable if and only if it is not mutable.

Each module has the potential of acting independently, so that there is natural concurrency in a system consisting of many modules. Since events happen instantaneously and the response of a module is not sensitive to anything but the sequence of events at the module, the event module implies concurrent interactions with a module cannot interfere with each other at the level of individual events. This non-

interference must be guaranteed by implementations which require a finite time interval to trigger the responses to an event. The response of a module is under the control of the designer.

In modeling concurrent systems it is sometimes necessary to specify atomic transactions. Atomic transactions are non-interruptible sequences of events at a module. Once a module starts an atomic transaction, it cannot accept any messages that are not part of the transaction until it is complete. Atomic transactions are sometimes needed to specify non-interference between concurrent sets of activities involving chains of multiple events at the same module. Atomic transactions must be used with care because they can lead to deadlocks if the protocols of the modules involved in a transaction are not compatible with each other, and can lead to starvation if a transaction goes on forever.

Modules can be used to model current and distributed systems, as well as systems consisting of a single sequential process. The event model helps to expose the parallelism inherent in a problem, because the only time orderings specified are those which are unavoidable and are agreed on by all observers.

Events can be triggered at absolute times. Such events are called temporal events. Temporal events are the means by which modules can initiate actions that are not direct responses to external stimuli. Formally a temporal event occurs when a module sends a message to itself at a time determined by its local clock. Unless explicitly stated otherwise, there may be an unpredictable delay between the time when the message is sent and the time when it is received, just like for any other message.

2. The SPEC Language.

The SPEC language uses second order logic for the precise definition of the desired behavior of modules. The Spec language provides a means for specifying the behavior of three different types of modules:

(1) Functions

(2) State machines

(3) Types

Each of these types of modules is described in the following pages along with examples of each type of module.

a. Functions.

Function modules are immutable and calculate functions on data types, where "function" is interpreted as in standard mathematics. Usually function modules provide only a single service and hence accept anonymous messages. Figure 2.10 gives an example of the specification for a square root function.

```
FUNCTION square_root(precision:real)
 WHERE precision > 0.0

 MESSAGE (x:real)
  WHEN x>= 0.0
  REPLY (y:real)
  WHERE y >= 0.0 & approximates (y*y,x)
 OTHERWISE REPLY EXCEPTION imaginary_square_root

 CONCEPT approximates (r1 r2:real)
  --True if r1 is a sufficiently accurate
  --approximating of r2.
  --The precision is relative rather than absolute
  VALUE (b:boolean)
   WHERE b<=> abs ((r1 - r2)/r2) <= precision
END

 Note: "--" introduces a comment and all keywords
     in Spec appear in all capital letters
```

Figure 2.10
Function Example.

b. State Machines.

A machine is a module with an internal state, i.e., machines are mutable modules. Figure 2.11 shows an example of a machine. The behavior of the machines is described in terms of a conceptual model of its state, rather than directly in terms of the messages that arrived in the past, because descriptions in terms of such a conceptual model are usually shorter and easier to read.

```
MACHINE inventory
 --assumes that shipping and supplier are other modules
STATE (stock:map(item,integer))
INVARIANT ALL (i:item::stock[1] >= 0)
INITIALLY ALL (i:item::stock[1] = 0)

MESSAGE receive (i:item,q:integer)
 --Process a shipment from a supplier.
 WHEN q > 0
  TRANSITION stock[1]=*stack[i] + q
   --Delayed responses to backorders are not shown.
 OTHERWISE REPLY EXCEPTION empty_shipment

MESSAGE order (io:item,qo:integer)
 --Process an order from a customer.
WHEN 0 < qo <= stock[io]
 SEND ship (is:item, qs:integer) TO shipping
  WHERE is = io, qs = qo
 TRANSITION stock[io] + qo = *stock[io]
WHEN 0 < qo > stock[io]
 SEND ship (is:item, qs:integer) TO shipping
  WHERE is = is, qs = stock[io]
 SEND back_order (ib:item, qb:integer) TO supplier
  WHERE ib = io, qb + qs = qo
 TRANSITION stock[io] = 0
OTHERWISE REPLY EXCEPTION empty_order
END
```

Figure 2.11
Machine Example.

c. Types

A type module defines an abstract data type. An abstract data type provides many services therefore the messages of a type module usually have a name. An abstract data type consists of a set of instances and a set of primitive operations involving the instances. The instances are the individual data objects belonging to the type. The instances of an abstract data type are black boxes. The properties of the instances are not visible directly, and can only be observed and influenced by means of the primitive

operations. The properties of an instance are determined by the primitive operation that created the instance and the sequence of primitive operations applied after it was created.

Data types are either mutable or immutable. For immutable types the set of instances and the properties of each instance are fixed. Operations producing instances of the type are viewed as selecting members of this fixed set. Figure 2.12 is an example of an immutable abstract data type.

```
TYPE rational
 INHERIT equality {rational}
 MODEL (num den:integer)
 INVARIANT ALL (r:rational::r.den ~= 0)

 MESSAGE ratio (num den:integer)
  WHEN den ~= 0
  REPLY (r:rational)
  WHERE r.num = num, r.den = den
 OTHERWISE REPLY EXCEPTION zero_denominator

 MESSAGE add (x,y:rational) OPERATOR +
 REPLY (r:rational)
 WHERE r.num = x.num*y.den+y.num*x.den,
     r.den = x.den*y.den

 MESSAGE multiply (x y:rational) OPERATOR *
 REPLY (r:rational)
 WHERE r.num = x.num*y.num, r.den = x.den*y.den

 MESSAGE equal (x y:rational) OPERATOR =
 REPLY (b:boolean)
 WHERE b <=> (x.num*y.den = y.num*x.den)
END
```

Figure 2.12
Immutable Abstract Data Type.

The state of a mutable data type consists of a set of instances which have internal states. The initial state of a mutable type is an empty set of instances. Mutable

types have operations for creating new instances, and usually also operations that can change the properties of an instance once it has been created. An example of a mutable abstract data type with immutable instances is the set of unique identifiers for the objects in a database.

An instance of a mutable data type is very similar to a state machine, except that the state machine is implicitly created at the start of the computation, while the instances of a mutable data type are created as a computation proceeds. A state machine has exactly one instance, while a mutable data type can have any number of instances. Figure 2.13 is an example of a specification of a mutable data type.

```
TYPE queue {t:type}
 INHERIT mutable {queue}
  --Inherit definitions of the concepts new and defined.
 MODEL (e:sequence)
  -The front of the queue is at the right end.
 INVARIANT tue
  --Any sequence is a valid model for a queue.

 MESSAGE create
  --A newly created empty queue.
 REPLY (q:queue{t}) WHERE q.e = []
 TRANSITION new(q)

 MESSAGE enqueue (x:t, q:queue{t})
  --Add x to the back of the queue.
 TRANSITION q.e = append([x], *q.e)

 MESSAGE dequeue (q:queue{t})
  --Remove and return the front element of the queue.
 WHEN not_empty (q)
  REPLY (x:t)
  TRANSITION *q.e = append (q.e,[x])
 OTHERWISE REPLY EXCEPTION queue_underflow

 MESSAGE not_empty (q:queue{t})
  --True if q is not empty.
 REPLY (b:boolean) WHERE b <=> (q.e ~= [])
END
```

Figure 2.13
Mutable Abstract Data Type.

III. SYSTEM DESIGN.

The first stage of the design effort was to analyze the requirements and obtain a better understanding of the SPEC language. To facilitate this process, a language reference manual [Ref. 19] was developed. This language reference manual describes many of the finer points in the SPEC language and provided a firm starting point for the type checker. It assisted in illuminating many of the issues that would have to be addressed in the design and capabilities that the type checker would have to encompass.

A. SPEC LANGUAGE TYPE CONSISTENCY CONSTRAINTS.

Like most computer languages, SPEC has many constraints on the naming and use of various operands. These constraints were derived from [Ref. 1] and [Ref. 19].

1. SPEC Language Semantic Issues.

a. Definitions.

- **Descendant of a Module**: A module is considered to be a descendant of another module if it explicitely inherits the traits of that module using an INHERIT clause.
- **Local**: A name is local if it is only visible to the module / entity in which it is contained.
- **Global**: A name is global if it is visible to the entire specification.
- **Signature**: The signature of a name is the ordered set consisting of the actual name, the arguments associated with that name and the types associated with the arguments.
- **Boolean Value**: A value that may only take on the logical values of "true" or "false".

- **Unique Definition Constraint:** Only one definition of a concept or message with the same signature is allowed to be visible in any of the modules in a well formed specification.

- **Definition Consistency Constraint:** Some concepts may have to be renamed before they can be imported or inherited.

- **Import Consistency Contraint:** A concept can be imported from another module only if the other module defines and EXPORT's the concept.

- **Instance Consistency Constraint:** Requires that the actual parameters of an instance of a generic module must satisfy any constraints mentioned in the WHERE clause after the generic parameter declaration.

- **Input Coverage Constraint:** Requires every concept to have proper values for all possible inputs satisfying the precondition. Also, the WHERE and TRANSITION clauses of each message must have proper values for all states and input values satisfying the associated preconditions.

- **Congruence Consistency Constraint:** A property of MESSAGES and CONCEPTS that is true if they mean the same thing for all equivalent conceptual representations.

- **Completed Specification:** A specification that meets all of the Constraints and scoping requirements of the SPEC language and contains no instances of the not yet defined clause ("?").

b. Scoping.

- The names of MODULES are global and unique. No module name may be redeclared at any other point in the specification.

- The names of MESSAGES and EXCEPTIONS are global.

- The names of CONCEPTS are local to the module in which they are defined. Concepts may be inherited by another module with the use of an INHERIT clause in that module. A Concept may only be associated with other modules if:

  (a) It is explicitly exported with an EXPORT clause and

  (b) It is explicitly imported into the module it is to be associated with by an IMPORT clause.

- The FORMAL PARAMETERS of a generic module are visible in the module in which these names are defined.

- The component names of the MODEL of a type are visible in the module in which the names are defined and in any descendants of that module.

- The component name of the STATE of a machine are visible in the module in which the names are defined and in any descendants of that module.

- The FORMAL PARAMETERS of a message are visible to the entire specification of that message.

- The FORMAL ARGUMENTS of a message are visible to the entire specification of that message.

- The FORMAL ARGUMENTS of a reply clause are visible from their declaration to the end of the when or otherwise clause in which they are declared. If no when or otherwise clause exists, they are visible until the end of the message specification.

- The FORMAL ARGUMENTS of a send clause are visible from their declaration to the end of the when or otherwise clause in which they are declared. If no when or otherwise clause exists, they are visible until the end of the message specification.

- The FORMAL ARGUMENTS of a generate clause are visible from their declaration to the end of the when or otherwise clause in which they are declared. If no when or otherwise clause exists, they are visible until the end of the message specification.

- The visibility of LOCAL VARIABLES declared in a CHOOSE clause extends from their declaration to the end of the when or otherwise clause in which they are declared. If no when or otherwise clause exists, they are visible until the end of the message specification.

- The scope of variables bound to a quantifier extends from the "(" following the name of the quantifier to the matching ")".

- All identifiers in SPEC must fall into one of the above categories.

c. Naming Constraints.

- The name of a module is considered unique if there is only one module defined with its given name.

- The name of a message is considered unique if there is only one occurrence of that name with it's specific signature within its scope.

- The operator of a message is considered unique if there is only one occurrence of that operator with the specific signature of its corresponding name within the operators scope.

- The name of a concept is considered unique if there is no other definition of that name with the same signature within the name'S scope.

- Any other name construct is considered unique if there is no other occurence of that name within its defined scope.

d. Type Consistency Constraints.

- An operation which is referenced to a specific module with the "@module" qualifier must be defined (or inherited by) the referenced module.

- If the "@module" qualifier is not used to clarify the use of an operator or message name, there must be exactly one candidate operation matching the types of the actual parameters.

- Arguments and Parameters in SPEC are specified by position. If a value or name is given for the arguments or parameters used in a call to a construct (the actual parameters), the types of the names or value must match the corresponding formal parameters or arguments.

- There must be a unique correspondence between the actual parameters and the formal parameters. For example, if the $ operator is used to specify a variable number of parameters in the formal definition, it must be determinable as to which actual parameters the $ will be bound.

- An expression following a WHERE clause must evaluate to a boolean value.

- An expression following a WHEN clause must evaluate to a boolean value.

- An expression following a SUCH THAT clause must evaluate to a boolean value.

- An expression following an IF or ELSE_IF clause must evaluate to a boolean value.

- The types of the expression following the "::" specification of a quantifier must match the requirements of the quantifier. Predefined constraints are:

ALL	Boolean
SOME	Boolean
NUMBER	Any type with an equality operator.
SUM	Any type with a commutative & associative "+" operation.
PRODUCT	Any type with a commutative & associative "*" operation.
UNION	Any type with a commutative & associative union.
INTERSECT.	Any type with a commutative & associative intersection.
MAXIMUM	Any type with a partial order "<=" operation.
MINIMUM	Any type with a partial order "<=" operation.

- The expressions on either side of a conditional operator must be of the same type.

- All of the normal REPLY clauses of the same message must be of the same type.

- All of the REPLY EXCEPTION clauses with the same exception condition in the same message must be of the same type.

- The definition of each message used in an expression must not contain any TRANSITION clauses.

- If a SPEC predefined operator is overloaded, the overloading message must have the same number of arguments as the defined operator in the SPEC library. For example, the "+" operator cannot be overloaded to a message that requires three arguments.

B. CONCEPTUAL MODEL.

1. Requirements.

The SPEC type consistency constraints identified many different requirements for the type checker. The more distinct requirements are:

- When a name is used in the specification, all defined argument lists for that name must be searched to determine the correct signature for that usage. If more than one possible matching signature is found, an error message must be reported listing all the conflicting usages. Figure 3.1 shows three skeleton modules. In the first two modules define two types, "nat" and "integer". They also define two messages, "add", each of which is a legal definition within its scope. The third module uses the "add" message. During the type checking process, an error must

be reported in function "does_something" stating that more than one possible signature match for the "add" message exists and reporting the conflicting bindings.

```
TYPE nat
  MODEL
  INVARIANT true

  MESSAGE add (n : nat, i : integer)
        REPLY (i2 : integer) WHERE i2 = i + n
END


TYPE integer
  MODEL
  INVARIANT true

  MESSAGE add (n : nat, i : integer)
        REPLY (i2 : integer) WHERE i2 = i + n
END


FUNCTION does_something
  MESSAGE (i : integer, n : nat)
        REPLY (i2 : integer) WHERE i2 = add(n,i)
END
```

Figure 3.1
Conflicting Name Bindings.

- A data structure must be available at all times which retains the names, signature, operator(s), module name, return type and parameters for each message in the specification.

- A data structure must be available during importation which retains the names, signature, module name, return type and parameters for each exported concept in the specification.

- During name analysis, all module names must be examined prior to the examination of any other name. The examination of message names, concept names, a module's formal parameters and the state or model clause variables should then be accomplished in order.

- Any variable names or types declared within any other SPEC structure are visible within that structure only subject to defined visibility rules.

- The type of every visible name must be immediately determinable during the type checking process to enable type consistency checking.
- Every name must be unique according to its scope and signature. If a name is not unique, an error must be reported.
- All the formal arguments and parameters of a name must be retained in full (i.e., the "type" and the name saved) in order to facilitate proper checking of variable argument or parameter lists. In this way, if the name is bound within the actual arguments or parameters, it is determinable which formal argument or parameter is associated with that name.

2. Model.

Based on these requirements, a design was developed that provided an efficient solution. The cornerstone of this design was the means in which a signature lookup was accomplished. The best solution this research found was to have a map from a name to a set of tuples. Each tuple in this set represents one distinct overloading of the name in the domain of the map. By searching this set of tuples, the specific overloading which is being used can be found.

To provide an efficient means for information lookup, each tuple in this set contains a list and a number. The list is an ordered list of tuples and each tuple in the list contains information on one of the formal arguments in the signature. The number is a value or cross reference that when "looked up" in the symbol table provides immediate access to all information concerning that symbol.

The tuple representing one of the formal arguments or formal parameters consists of two elements--the name of that element and its "type". The "type" that is placed in this second element is derived from a map which has a domain consisting of all the types that are visible at that point and a range consisting of a translated text for that specific

type. The translated text is simply a modified version of the actual type name. If a type name belongs to a concept, it is local to the current module, so an "@" symbol is appended to the name followed by the current module's name. If the type name belongs to a module, the range matches the domain. In this way, if a concept is defined differently in two different modules the "relationships" (e.g., messages, etc.) between the modules must use the concept they were defined with and not the corresponding concept in the other module. In Figure 3.2 the two types of entries permitted in the "type" map and the module that defines them are shown.

```
TYPE complex
   MODEL ( re : real, im : imaginary_part)
   INVARIANT

   CONCEPT imaginary_part : type
         WHERE imaginary_part = real
END
```

Actual Name	Translated equivalent.
complex	complex
imaginary_part	imaginary_part@complex

Figure 3.2
Limiting Type Visibility.

Based on these features, the symbol table becomes a map from the cross reference value to a tuple. This tuple contains the required information for each symbol, its parameters, class, textual name and type. The parameters element is a tuple which represents the formal parameters (if any) of the symbol. The class element contains some

representation of the class (function, message, concept, etc.) that the symbol belongs to, and the textual name element contains the actual text of the symbol (used for error reporting).

While determining the conceptual model of the last element of this tuple, it was noted that each symbol that would be placed in the symbol table was either a variable, a concept or message or a module name. Interestingly enough, this indicated that the type element of the tuple contained in the domain could have a dual purpose. If the symbol was a variable or "non-function" concept (a concept without a VALUE clause), the actual type of that name could be placed in that field. If the symbol is a message or concept with a VALUE clause, the type that the symbol returns could be placed in that field; and if the symbol was a module name, no information needed to be placed in that field.

Actually building these tables presented another problem. Due to the declaration requirement that a module name could not be redeclared and that concept and message names are visible throughout the entire module they are defined in, the necessary "name" table has to be built in three "layers". In the first layer, all of the module names are collected into a table and any redeclarations are identified. These module names are then passed down into the second layer during which all message and concept names are added to the table. Additionally, if a message has an operator associated with it, the operator can be treated as a name unto itself, with the same arguments as the message and stored in the table accordingly.

When this table is returned to the top of the semantic tree, one additional level of indirection is added to it so that a name declared in one module doesn't "overwrite" the

identical name declared in a different module. This level of indirection is added by taking the original map and making it the range of a new map whose domain is the module name. Figure 3.3 shows this process.

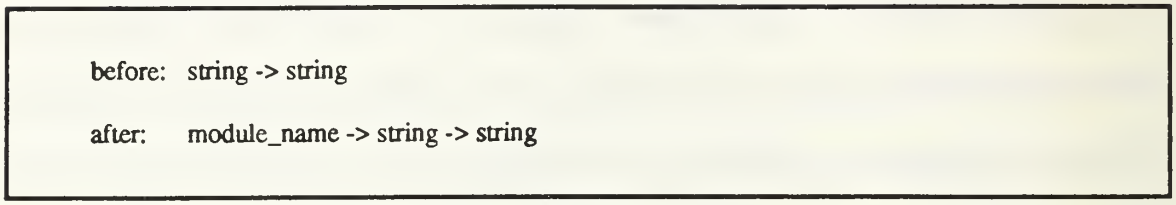

Figure 3.3
Localizing a Map.

The final layer in the name analysis process takes this table produced by the second layer and "cuts" it within each module so that only those names defined in that module are passed back down. All other names that are encountered within that module are then added to it, according to the scope rules of the SPEC language. With the tables being "manuevered" through the semantic tree during this layer, the type consistency analysis can be performed. Additionally, if the tables produced by the second layer are passed down the tree also, these tables can be used to verify whether a message exists or doesn't exist in another module. Figure 3.4 demonstrates this name layering process.

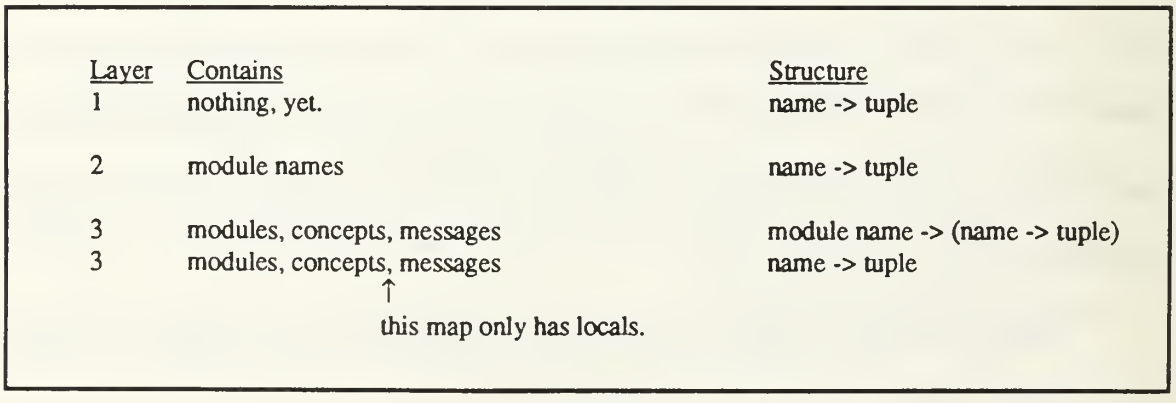

Figure 3.4
Name Layering.

With these tables, the type consistency analysis simplifies to two distinct parts. The first part involves obtaining the correct symbol from the tables produced in the layering process. To determine that only one unique possibility exists for this symbol, both of the layer three tables must be searched--the local table and the table which contains all the symbols defined in other modules. However, in the second table, only messages need be examined.

The second part of the type consistency analysis involves checking the actual type of the symbol. Ideally, this should only be a "lookup" in the symbol table, but since a message or concept may have a value that is transitively dependent on another message or concept,a routine that recursively resolves the type must be performed. Figure 3.5 shows a two dimensional transitive dependency situation. The first message, message_1, has a resultant value that is dependent upon a concept, dimension_2. Theoretically, this transitivity could be repeated extensively.

```
MESSAGE message_1 ( ... )
  REPLY dimension_2 ( ... )

CONCEPT dimension_2 ( ... )
  VALUE ( ... )
```

Figure 3.5
Result Values Transitive Dependency.

To conclude the process, the type is passed up to the next higher level of the semantic tree and used to resolve that level. Additionally, any errors encountered are concatenated onto the error messages from the "children" of the current level and also

passed up. When the uppermost level of the semantic tree receives these error messages from its children, the type checking process is completed and the errors can be reported to the user.

C. DESIGN CONVENTIONS.

In an effort to increase the readability of the source code, it was decided that the M4 macro preprocessor would be used and a stardardized attribute naming schema adopted. The attribute naming schema assisted in cutting down the source code size, but the M4 macro preprocessor helped significantly more. The M4 preprocessor "shrunk" the actual code size almost 50% (3926 lines prior to expansion, 7591 after) by coalescing multiple source code lines into one line of M4 code.

1. Attribute Naming.

The primary rule followed in the naming of all attributes was to make the name as descriptive as possible concerning the purpose of the attribute, without "exploding" the size of the source code. Additionally, each attribute is appended with an underbar (_) followed by a descriptive character (s or i) which signifies the use of the attribute (synthesized or inherited). Some sample names include "module_name_s", "visible_types_i" and "ip_stbl_s". A complete listing of all descriptive names is contained in Appendix D.

Certain abbreviations were adopted to assist in the naming, without making the name too long. Some of the more common abbreviations include:

ip : signifies that the attribute is currently "in the process" of being built. The information currently contained in this attribute is not reliable for any other purpose than building the final attribute and thus should not be used for any other purpose.

lclzd : denotes that a table is "localized". A localized table is normally a map with a domain consisting of a string and the range containing another map. The string in the domain string will always be a module name or a scope related value (such as "GLOBAL_TYPE_NAMES" in myconst.m4).

stbl : symbol table. Any attribute name prefixed by this word denotes an attribute that is part of the symbol table group of attributes (e.g., stbl_names).

xref : cross reference. Any attribute containing this abbreviation has a range which contains cross reference information in it (normally within a tuple). Many times, a prefix is appended to this word (mxref or mcmxref) to assist in the distinction of the attributes' purpose. Two of the more common attributes using this abbreviation are mxref (module cross reference) and mcmxref (module-concept-message cross reference).

env : environment. This abbreviation is commonly used in attributes that are passed down to non-terminals to "inform" the non-terminal of the environment within which it is currently being utilized.

2. M4 Macro Abstractions.

The actual M4 macro definitions are contained in three different files--"attrib_psg.m4", "mymac.m4" and "myconst.m4". Two additional M4 files were used in

this design (head.m4 and tail.m4) but they simply contain certain M4 commands that are required so that M4 will function properly with the Kodiyak tool. All of the M4 files are enumerated in Appendix 2.

a. Attrib_psg.m4.

This file contains all the M4 macros that are associated with a general attribute passing capability. They could be used in any Kodiyak program and are not at all specific to the type checker. Most of these macros have been derived from the "macros.m4" file developed by Robert Herndon and promulgated with the Kodiyak compiler. Some modifications were made to the actual definitions for the purpose of standardization, however. Most of these modifications involved changing a macro's name to more accurately reflect how many attributes were being passed and how many non-terminals these attributes are passed to.

There are six actual groups of macros within this file. Each group is characterized by a descriptive name, followed by two integer values separated by an underbar. The name details the purpose of the macro and the integer values represent the number of non-terminals being passed followed by the number of attributes affected (e.g., passio2_4--pass in order to two non-terminals, four attributes). The arguments for the macro follow in the same order as the integer numbers--non-terminals first, then attributes. The six names used for these macros are:

Passup : Pass up an attribute from a child non-terminal ($1, etc.) to the parent non-terminal ($$). Figure 3.6 graphically depicts this class.

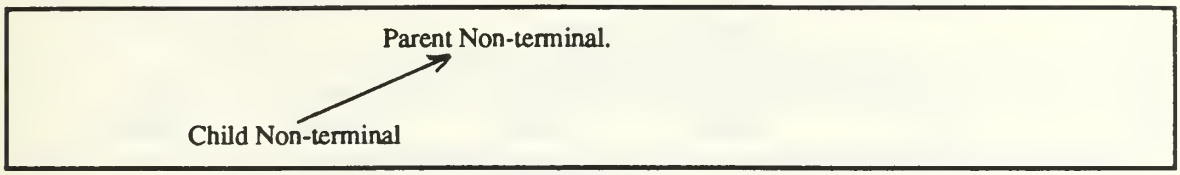

Figure 3.6
Passing Up an Attribute.

Passdn : As shown in Figure 3.7, an attribute is passed down from the parent non-terminal to child non-terminal(s).

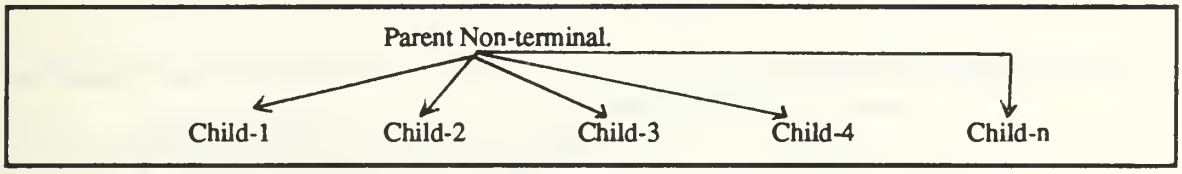

Figure 3.7
Passing Down an Attribute to "n" Non-terminals.

Passovr : Figure 3.8 shows how this macro "passes over" an attribute from one non-terminal to another. There are two variations of the passovr macro. The first variation is simply a "passovr" from one non-terminal to another (passovr_1). The second, which is a logical extension of the first, passes the specified number of attributes to more than one non-terminal from only a single non-terminal. To vividly display this significant difference from the weave and passio macros, the naming of this variation is slightly different from the standard naming. An "x" was placed between the first integer (signifying number of non-terminals) and the underbar. The "x" is best interpreted as a "times". Thus, the macro looks like "passovr2x_1" which means "passovr two times, one attribute".

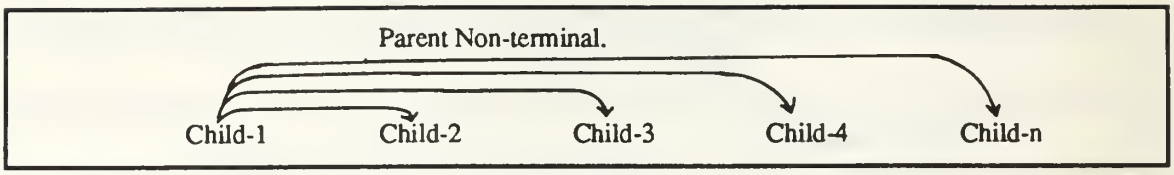

Figure 3.8
Passing Over an Attribute to "n" Non-terminals.

Passio : Pass an attribute in order from the parent non-terminal, through the specified children non-terminals and back to the parent. Figure 3.9 shows this commonly used macro.

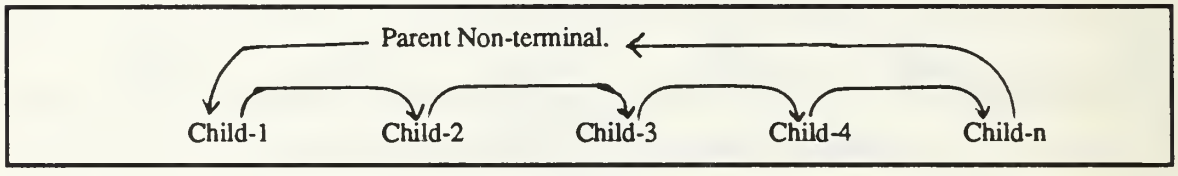

Figure 3.9
Passing an Attribute in order to "n" Non-terminals.

Weave : Weave an attribute from a child non-terminal, through other specified children non-terminals and into a non-terminal. As shown in Figure 3.10, this macro is similar to Passio, except the parent non-terminal is not affected.

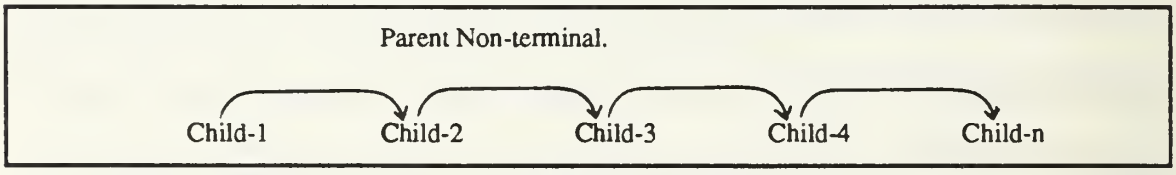

Figure 3.10
Weaving an Attribute to "n" Non-terminals.

cat...up : Concatenate "..." up to the parent non-terminal from the children. The "..." may be either the abbreviation "str" meaning string or "map" meaning map.

b. Mymac.m4.

This file contains macro definitions that are unique to the type checker. Each of these definitions provides a shorthand method of expressing multiple lines of Kodiyak code and greatly simplifies the readability of the source. They can be logically divided into three groups.

(1) Declaration Group.

The declaration group consists of four different macros, which are used in the attribute definition section of the Kodiyak source. They are designed to make each non-terminal's defined attribute list more readable.

(2) Symbol and Visibility Tables Group.

These macros are defined to assist in the attribute evaluation section of the Kodiyak source. They primarily provide simple statements for passing the symbol (or visibility) tables down from one non-terminal to another.

(3) Attribute Evaluation Group.

This group of macros is also used in the attribute evaluation section of the Kodiyak source. They simplify the amount of code used to express the equations to "make a declaration", etc.

c. Myconst.m4.

This M4 definition file contains the various symbolic constants used throughout the Kodiyak source. Some slight variations of these constants are also used in the C language file mylib.c and the correlation between them is vital to the type checker. All relationships are detailed as comments in the mylib.c file.

IV. IMPLEMENTATION.

A. SEMANTIC INFORMATION STORAGE STRUCTURES.

To properly type check the SPEC source code, various tables were required. These tables contained information relevant to each module, such as message names, arguments, parameters and result types. The primary requirement that necessitated the use of these tables was the "non-block" structured nature of certain SPEC constructs. For example, when the information regarding a specific message is looked up, a "match" must be searched for in the current module and all type modules corresponding to an argument type of the message. These type modules may or may not have been previously declared. Another of the non-block structured SPEC structures is the fact that concept names are visible throughout the entire module in which they are enclosed, thus requiring, at the very least, that the module be "passed over" twice--once to obtain the information about concepts, and the second time to type check the rest of the module.

1. Module Types.

The module types table contains a listing of all the valid type names that are visible in a module that must be accessible immediately upon entering the module (e.g., concepts and module types). This table is especially important due to the fact that all other tables depend upon it. Whenever a type is stored, its "translation" in this table is

used, vice its actual name. The translation represents a globally unique name for the type. This permits the localizing of types that are truly local to the module such as concepts.

The table is structured as a map from strings to a map of strings to strings. The domain string consists of the module name, the domain string of the range map containing the actual name of the type (e.g., real) and the range string containing the translation that will be used to reference the type. To symbolize local names, the current module name is prefixed by an "@" symbol and the actual local name (local_name@module_name).

One of the most important uses of this table is to select appropriate portions (or "cuts") from it when a module is entered, and place these "cuts" in the visible types table. This table is then used throughout the module.

2. Symbol Table.

Due to the lack of tuple structures in the Kodiyak language, the symbol table actually consisted of five different tables. Each of these tables has a unique purpose. The primary table, called the symbol table, consists of a map from strings to a map consisting of strings to strings. The primary domain of the map contains module names. The domain of the "range map" consists of the symbol's name and the range of this map contains a group (or variant tuple) of patterns. Each pattern is separated by a delimiter (PATTERN_DELIMITER).

A pattern is a tuple consisting of a variant sized tuple of formal or actual arguments, and a cross reference value. Each element in the formal or actual arguments

"subtuple" is separated by a delimiter (ELEMENT_DELIMITER) and this subtuple is separated from the cross reference value by another delimiter (XREF_DELIMITER). Figure 4.1 shows the format for a pattern and a pattern string. In this Figure, the ELEMENT_DELIMITER is represented by •, the XREF_DELIMITER by ♣ and the PATTERN_DELIMITER by ♦.

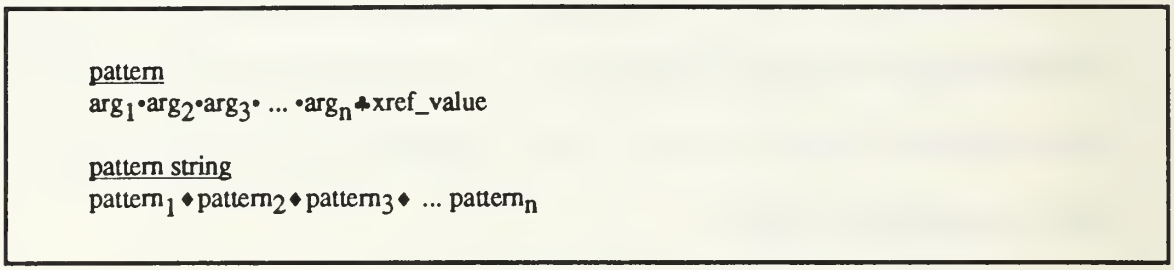

Figure 4.1
Example of a Pattern

The cross reference value is of extreme importance to the type checker. It is used in the rest of the maps which contain the symbol table information to access the information. Without this cross reference value, much of the information required could not be accessed properly.

a. Textual Names.

This is the first table containing the symbol table information, other than the actual symbol table. Its structure is a map from integers to strings. The domain of the map contains the cross reference value and the range contains the actual text of the name of that symbol. This information is commonly used in error messages.

b. Parameters.

This table contains the formal parameters associated with a symbol. Its structure is a map with a domain of integers and a range of strings. The domain is the

cross reference value of the symbol and the range is a string consisting of a concatenation of all the parameters of the symbol and their types. Each element in this concatenation is separated by a delimiter and within each element, the element's name and its type are separated by a different delimiter.

c. Results.

The results table is a map from an integer cross reference value to a string which contains the result type of the symbol. A result type can be interpreted in two different ways. In the case of a message or concept with a VALUE clause, the result type contains the type of the reply or value. If the symbol is a variable or variable-type concept, the result type contains the type of the variable.

Since the result table must be built prior to its use in the actual type consistency checking, if the type to be placed in the table is a message or concept call, the actual text of the message or concept call is prefixed by a special symbol, which I call the reference symbol, and placed in the range of the map. This presents the requirement that a C language function be used to assist in resolving the type of any construct, since this result table value may be transitively dependent on other values. This C language routine, which I have named "Resolve_Type", will recursively analyze the result types of different symbols until a result type is found that is not preceeded by the reference symbol.

d. Classes.

This table is used to uniquely identify the classes of various names. It is extremely important and allows checking of a name to determine if it is a message,

module, concept or variable name. Each different class has a unique value and these values are detailed in the file "myconst.m4", which is listed in Appendix B.

3. Visibility Tables.

The two visibility tables used by the type checker address the block structured constructs in the SPEC language. They are initially constructed in a module's interface and then passed into the various other parts of the module. These tables are then added to and passed into additional non-terminals based on the scoping rules of the variables in SPEC.

a. Visible Types.

The visible types table is constructed initially from the module types table. In the module's interface, all types corresponding to the module's name are extracted from the module types table and placed in the visible types table. This table is then passed into all the other non-terminals in the program, as dictated by the scope rules. The visible types table is not added to by concept names since these names are already in the table. Any other types that are declared (in variable names) are added to the table. This table is used to build any table requiring a type. The value of this type could be localized or the actual name as discussed above.

b. Visible Names.

The visible names table is initially formed in the interface section of a module. Currently, it is primarily used in the name declaration routines to determine if a name is already declared. It will also be used extensively in the type checking routines to

obtain the cross reference value of a symbol when that symbol is used. As done in the visible types table, it is passed to all non-terminals as dictated by the scoping rules of SPEC.

B. MAJOR ATTRIBUTES.

1. Error Reporting.

Errors are reported in SPEC in one of two ways. If the error can be identified by the attribute grammar equations, a call to the C language function "error_message" is made. This function returns a string in the correct format for an error message and contains the appropriate information. If the error cannot be identified by the attribute grammar equations, but can be identified by a "C" routine, that C language routine may call the error_message function directly.

a. verror_message.

The verror_message function is the actual C language function corresponding to a call to "error_message" in the attribute grammar equations. It is contained in the mylib.c file and detailed in Appendix B. All of the codes for error messages are a constant integer value and are detailed in the file myconst.m4 and mylib.c. Each error has a unique code. Each code has a predetermined number of arguments that it requires to properly report the error. When the error in "invoked", all of these arguments must be passed to the verror_message function in the correct order.

b. Declaration Errors.

Declaration errors are determined by two different C language functions--"vcheck_simple_decl" and "vcheck_complex_decl". The first of these routines,

check_simple_decl, is used to check all declarations that do not have a signature of arguments associated with them. The second routine, check_complex_decl, is used for declarations that have a signature. Each of these functions returns a null string if the declaration is not previously defined or an error message otherwise. All of the declaration errors are placed in an attribute named "d_error_s". Prior to adding the current declaration to the respective table, this attribute is checked and if it is NULL, the attribute is added.

c. Error Concatenation.

All errors are passed up the semantic tree in an attribute named error_msgs_s. Each non-terminal in the tree has this attribute associated with it. At each level, the attributes are concatenated with the lower levels and any errors that were discovered in that level and passed up to the higher levels.

2. Building the Symbol Table.

As mentioned above, the symbol table actually consists of five structures, each structure having a unique purpose. It was determined that the four secondary tables (textual names, parameters, results and classes) could be built independent of the primary table (stbl) since these tables depend only on the cross reference value which can be determined immediately.

The primary table (stbl) is built in two layers, due to the declaration precedences of SPEC. These precedences require that module names be globally visible and unique, and messages and concepts be unique within their module. Therefore, the first layer of the symbol table building process collects all the module names and passes them up to

the root non-terminal (in the "ip_mxref" attribute). These names are then passed back down the semantic tree (as the ip_mcmxref attribute) and all the message and concept names are collected in accordance with the scope rules of SPEC. In this way, redeclarations are reported in a logical, semantically correct order. For example, if a message or concept name redefines a module name, an error is reported when the attempt is made to define the concept or message name. After this primary table is built, it is passed down into all non-terminals and used to construct the visible names table in the interface section of each module.

The other four parts of the symbol table are built in one layer. This layer collects all the values and their appropriate range and passes the results up so that they can then be passed back down and used by all the non-terminals.

3. Extended Types.

Due to the need to store result types in a result type table, it was necessary to develop a slightly modified type called extended type. If a type is immediately determinable, such as a literal or type name, the range of the visible_types table for the actual type name is placed in the "xten_type" attribute. If the type of the construct is not immediately determinable, the actual text of the construct is placed in the attribute. In this way, a C language function, resolve_type, can take this value and resolve the type of the construct or symbol when needed.

C. NAME ANALYSIS.

Name analysis is the first of three aspects in the type checking process. During name analysis, tables are built reflecting all the names used in the SPEC code. If an invalid

declaration is attempted or an invalid type used, an error is reported. The tables built during name analysis are used during the second aspect (type consistency checking) to determine if any errors occur.

1. Checking if an identical declaration exists.

There are two routines used to check if a declaration exists prior to declaring a name--check_complex_decl and check_simple_decl. Each of these routines takes a signature and analyzes all other symbols in the current scope to see if that name has been previously declared. If it has, they will report an error, otherwise, they return a NULL string.

2. Making a new declaration.

To make a declaration, three macros were defined in the "mymac.m4" file. Only one of these macros need be used. Each of them checks a string type attribute (always named d_error_s) and if that attribute is NULL, makes the declaration. If the attribute is not NULL, meaning that the declaration would be a "redeclaration", the declaration is not made.

3. Reporting an error.

An error is reported at the declaration point as detailed in section 1 above. In each non-terminal structure that declares a new name, the attribute "d_error_s" is concatenated with all other error messages from the children non-terminals and the result is passed up the semantic tree. In this way, the errors encountered are placed in the correct position within the list of error messages.

D. IDENTIFYING ERRORS TO THE USER.

The third and final aspect of type checking reports any errors that occured to the user. Currently all errors are identified by SPEC source line numbers. If no syntax errors occur, these error messages are output to the standard output at the end of program execution. Currently, the SPEC grammar does not have syntactic error productions added in, although they have been developed for previous versions of the grammar.

V. EXTENSIONS.

A. TYPE CONSISTENCY ANALYSIS.

Type Consistency analysis is the second aspect of the type checking process. Although the C language routines were coded and syntactically debugged, the required attributes have not been implemented into the Kodiyak source code.

1. Seeking the Correct Symbol Table Entry.

The process of finding the correct symbol table entry is similar to that of checking to see if a declaration has been made, except for the fact that actual arguments instead of formal arguments are included in the "source" name. To obtain the correct symbol table entry, a call is made to the C language routine "seek_symbol". This routine will search the current environment (visible_names), and the global environment (stbl) to determine if the name exists. If more than one possible interpretation of the name exists, the function will return the appropriate error message, listing all possible interpretations. If a unique candidate exists for the signature, the string representation of the cross reference value of the symbol is returned. Conversely, if no symbol could be found that matches the signature passed to seek_symbol, the string representation of the integer value "0" will be returned. If desired, this routine could be easily modified to allow an error message to be returned if no symbol exists.

2. Obtaining a Symbol Table entry's type.

If a symbol has been found that matches the actual name of the symbol, another "C" language routine, "resolve_type" is called to obtain the result type of the symbol. Using the information provided in the "stbl_results" table, this routine recursively analyzes the symbol's value until a valid type name is obtained. The recursive analysis is required to resolve this table's transitive dependency on other messages or concepts as discussed in Chapter 3. When this transitive dependency is resolved, the type name's translation in the "visible_types" table is returned to the Kodiyak attribute. This attribute is then passed up the semantic tree and used at "parent levels" to determine if an operation is valid.

3. Determining if an Operator is defined.

During the name analysis, an entry was made in the symbol table for each operator overloading. Since SPEC is entirely defined in terms of the standard type library, by processing the standard type library together with the SPEC code to be type checked, all possible operator meanings are placed in the symbol table. To determine if an operator use is valid, simply take the operator's textual representation, append the appropriate arguments (determined by its use) to it and use the routine "seek_symbol". This routine will then return the cross reference of the message that overloaded that operator. Then the result type may be obtained as discussed in section 2 above.

4. Reporting Errors.

The reporting of type checking errors is very similar to the reporting of declaration errors with one small exception. Since "seek_symbol" always returns a string

value, the attribute in which this value is stored must be checked to see if the attribute contains an error message or the string representation of an integer (greater than 0). If the attribute contains an error message, that error message is then concatenated with the error messages generated by the children non-terminals and the result passed up the semantic tree. Otherwise, only the error messages generated by the children are concatenated and passed up the tree.

B. SPECIAL SPEC LANGUAGE ISSUES.

Some of the more complex SPEC issues such as inheritance, instance declarations and importation/exportation were addressed and accounted for in the design, but not implemented. The proposed methods for implementing these features, based on the design is discussed below.

1. Inheritance & Instance Declarations.

Inheritance and instantiation present unique challenges to the generation of a type checker using an attribute grammar tool. One of the most significant problems arises because of the possible transitivity of either of these structures. For example, a module may inherit a module which inherits another module, which inherits another module, etc. This requires that the module which is the "lowest common denominator" be expanded first, then the next, etc.

Additionally, the way that inheritance is defined in SPEC poses other problems. Specifically, if a module inherits another module which contains a message with the same signature as the current module, the two messages are combined according to predetermined rules [Ref. 20] to form the expanded, resultant module.

a. Preprocessor Usage.

To address these unique problems, this thesis proposes the use of a preprocessor. This preprocessor would take the SPEC source code, expand it as necessary and present its output to the type checker. The type checker would then operate upon this intermediate source code and produce its error messages. If the "inheritance / instantiation tool" recognized any errors such as a circular inheritance, it would report these errors to the user and terminate.

b. Error Reporting Drawback.

The preprocessor would present to the type checker a modified version of the source code with no inheritance or instantiation (and probably write its output to a file). This however, presents a problem. Since the type checker reports error messages based on a source code line number, any errors identified would be associated with a line number relating to the "expanded" source, not the original SPEC source code.

c. Potential Advantages.

This methodology may have its advantages, however. If the process of inheritance introduces a structure that has semantic errors in it, the error could be looked up in the output of the inheritance tool and traced back to its originator. Also, with the advent of sophisticated text processing tools for the SPEC language, it may be possible to edit the "true source" code in one window while viewing the error in another window. Another implicit benefit may be that software developers could use the inheritance tool independently to examine the expanded specification to determine if they have "hidden" or "renamed" everything as appropriate.

2. Importation & Exportation.

One of the final issues addressed in the design was importation and exportation. Although they are two different constructs in SPEC, they are uniquely related--a concept may not be imported unless it is exported by the module in which it is defined. Additionally, importation and exportation do not present any of the problems posed by inheritance. They are not transitive and if a concept is already defined with an identical signature, the new concept cannot be imported and an error should be reported.

The way in which the design was built presents a simple solution to importation/exportation. Initially, a new structure (lclzd_exportables) must be built. This structure should be a localized map with a domain of strings and a range which is a map. This domain string would contain the name of the module as in all localized maps. The map which makes up the range should be a map from string to integer. It would contain as a domain the name of the concept, and as a range an integer value (0 or 1) representing the boolean value true or false. The range would be true if the concept is exported, false otherwise.

The second part of the solution is when an importation is requested by a module, this new table (lclzd_exportables) is checked immediately. If the module does not export the desired concept, an error should be reported. Conversely, if the desired concept is exported, the "visible_names" table would be augmented with the signature and cross-reference information of the concept(s). This augmentation process would require a C language function which processes the domain of the "stbl" structure for the specified module and name and then returns a string consisting of all the new patterns (signature

and cross reference information) which are to be added. This routine would have to be passed the "stbl_classes" structure so that it could verify that the symbols that it returns are indeed concepts and not messages.

C. IMPROVED ERROR REPORTING.

The type checker currently reports declaration errors in a way that is easy to understand, but sometimes difficult to find the conflicting declaration. In order to provide better feedback to the user, additional tables could be added to the symbol table to promulgate information that would assist in error reporting. For example, a table with a domain containing the cross reference value and a range containing the line number where that symbol was declared would enable error reporting to report the location of a conflicting declaration.

Another error reporting difficulty is that some of the SPEC constructs have WHERE clauses that require dynamic (run-time) evaluation. Although it is not feasible to automatically check these clauses, it is recommended that a warning message or pragma be output listing the where clause's contents so that the user could examine this to ensure the validity of the specification.

D. SUBTYPES.

Subtypes in SPEC are defined as concepts and have a WHERE clause associating the concept name and another defined type. They present a slight problem because, like inheritance or instantiation, a subtype may be transitively dependent on another subtype. The solution to this problem involves using two C language routines, one for declaring a

subtype and one for analyzing it. The first routine, "declare_subtype" would take the subtype name and the type which it is a descendant of and place it in a table. The second routine, "is_subtype" would then take a type name and recursively analyze this table, returning a boolean value representing the validity of the subtype. This routine "is_subtype" could then be used in some of the existing C language routines such as "type_equivalent" to assist in the determination of type conflicts.

E. VARIABLE ARGUMENT OR PARAMETER LISTS.

The implementation of variable argument or parameter lists (list preceded by a '$') is an interesting proposition. Since there are many diffent ways in which these lists may be fitted together, a recursive analysis is required. Currently, variable argument lists have been acknowledged, but the required recursive analysis has not been implemented. This analysis should take place in the routines that check a declaration and seek a cross reference.

VI. CONCLUSIONS.

A. INTEGRATION INTO A PROGRAMMING ENVIRONMENT.

To truly provide the SPEC user with an effective software development tool, the type checker must be integrated into a programming environment. In addition to a type checker, this environment should contain at least a syntax directed editor, pretty printer, test case generator and eventually a translator that will translate a significant part of the specification into a compilable target language.

In the short term, the type checker, syntax checker, inheritance preprocessor and pretty printer should be able to work together in a way that makes the actual separation of these tools transparent to the user. This could be accomplished efficiently by writing a unix command script that begins a tool execution when the previously running tool (if any) completes. In this script, the syntax checker, inheritance preprocessor and type checker should be called in sequence to provide the user with syntactic and semantic information concerning their program. Additionally, at least two options should be provided with this script. One option would allow the user to retain a copy of the file containing the specification after it has been expanded by the inheritance preprocessor and the other would run the pretty printer on the code if it is semantically and syntactically correct.

B. EVALUATION OF THE TYPE CHECKER.

Although the type checker is not as yet a usable tool, its feasibility has been researched and a solid groundwork has been laid for the rest of the implementation.

1. Kodiyak Deficiencies.

While researching this thesis, many deficiencies and "bugs" were found in Kodiyak. The primary deficiency was the lack of any types other than integer and string. The implementation of the type checker was forced to use many identical data structures for similar purposes that should have been one structure. Specifically, the symbol table required by the SPEC language necessitates the use of a tuple in the range. Since there is no tuple type in Kodiyak, four maps were used to contain the information.

The inability in Kodiyak to declare a global variable also presented a problem. Ideally, since the symbol table is not modified once it is built, it would be convenient (and conserve memory space) if this table could be placed in a variable or data structure that could be referenced from every production. In this way, fewer attributes would have to be passed down the semantic tree and the number of attribute equations would be decreased.

The lack of documentation in the Kodiyak C library is a substantial drawback. Since Kodiyak is entirely implemented in terms of other tools, the handling of strings and integers is defined in the C language and utilized by the Kodiyak processor as function or procedure calls. To effectively extend Kodiyak so that it could meet the requirements dictated by SPEC, many long hours of deciphering the source code and experimenting was required.

The lack of any predefined Kodiyak functions to output the contents of an entire map is a handicap. During the incremental implementation of the various maps that make up the symbol table, it was necessary to output the information they contained to verify the functioning of the attributes. Unfortunately, the only way to accomplish this task was to select each individual map entry and display it. After a short while, this became very tedious and so routines were built and debugged that dump one dimensional maps.

2. Kodiyak Benefits.

Probably the most beneficial feature of Kodiyak is its ability to preprocess M4 files prior to conducting the Kodiyak scan. When the preprocessor is extensively used and considered throughout the implementation, the source code size can be shrunk dramatically, making both the programmer's and reader's job easier. Additionally, the M4 macros defined by Robert Herndon proved to be invaluable.

Another positive feature of Kodiyak is the way it integrates the functioning of Lex, Yacc and the C Compiler to produce an executable product. Since all of these tools are reasonably well understood, many of Kodiyak's functions can be analyzed from another perspective, providing an alternative approach to debugging.

Kodiyak's C language interfacing ability, although difficult to decipher initially, proved to be a benefit in the long run. It provided a way to "work around" the deficiencies and implement the type checker in an efficient, sensible manner.

C. FUTURE WORK.

1. Extensions of the current implementation.

The type checker is feasible and worthwhile to complete. The extensions still required are implementable by two students, working independently and present no significant problems. One student should focus effort on the preprocessor and another on implementing type consistency checking, importation and integrating a current version of the error productions into the type checker.

Since the design and implementation of this thesis, the meaning of a signature has been extended to include the formal parameters of modules, concepts and messages. To implement this feature, the type checker's implementation of a "pattern" must be extended to include formal parameters by adding in a new delimiter and the additional information. All of the C language routines which check declarations and look up names must also be extended accordingly.

2. Incremental Type Checking.

One significant project that should be addressed in the future is the incremental type checking of the SPEC grammar within a syntax directed editor. This would then allow any errors to be identified concurrently with the writing of the specification, permitting better time utilization. Additionally, the benefit for individuals learning the SPEC language would be significant since as syntax or semantic errors were made, the reason and cause would be displayed immediately. A syntax directed editor for SPEC currently exists [Ref. 21].

D. GUIDELINES FOR EXTENDING KODIYAK.

The Kodiyak language is very simple to extend when the interactions between the C library and the actual Kodiyak tool are understood. These interactions are manifested by calls to functions in the C library which are built through strings in the actual Kodiyak AG code. For example, a Kodiyak language map reference translates into a call to one of six map lookup functions, depending on the domain and range of the map. Some guidelines for using the C language escape feature of Kodiyak are:

- Whenever a string is used directly from the Kodiyak program, the string should be immediately "flattened" to the temporary work area (by means of a call to xtstrflatten) and then IMMEDIATELY copied to a work area belonging to your routines. Leaving a string in the Kodiyak temporary work area can be fatal since Kodiyak overwrites that work area frequently.

- Always build in extensive error checking in your routines to avoid errors such as array overflow. "Silent" errors in your routines may cause other, reportable errors within Kodiyak which may confuse the situation.

- Syntactically debug your routines independently from Kodiyak (as best as possible) to avoid unnecessary (and frustrating) delays. The Kodiyak compilation process is not fast by any means--especially the C compilation phase.

- The Kodiyak program prepends a "w" to the name of a routine beginning with a "%" (e.g. a procedure) and a "v" to the name of any function before calling that function in C.

APPENDIX A - SPEC GRAMMAR.

This Appendix contains the version of the SPEC grammar used to implement the type checker. This version does not contain any of the syntactic error productions which have been developed or any attribute definitions. It is primarily provided as a quick reference for the grammar of the SPEC language and for contrast with the type checkers attribute grammar code which is contained in Appendix B.

```
! version stamp $Header: spec.k,v 1.10 89/02/11 20:11:31 berzins Locked $
! Kopas Version -- Updated to version 1.11 of grammar 20 April 89.
! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%define     Digit                                          :[0-9]
%define     Int                                            :{Digit}+
%define     Letter                                         :[a-zA-Z]
%define     Alpha                                          :({Letter}|{Digit}|"_")
%define     Blank                                          :[ \t\n]
%define Quote                                              :["]
%define Backslash                                          :"\\"
%define     Char              :([^"\\]|{Backslash}{Quote}|{Backslash}{Backslash})

! definitions of white space and comments

                                                           :{Blank}+
                                                           :"--".*"\n"

! definitions of compound symbols and keywords

AND                                                        :"&"
OR                                                         :"|"
NOT                                                        :"~"
IMPLIES                                                    :"=>"
IFF                                                        :"<=>"

LE                                                         :"<="
GE                                                         :">="
```

```
NE                                  :"~="
NLT                                 :"~<"
NGT                                 :"~>"
NLE                                 :"~<="
NGE                                 :"~>="
EQV                                 :"=="
NEQV                                :"~=="

RANGE                               :".."
APPEND                              :"||"
MOD                                 :{Backslash}|MOD
EXP                                 :"**"

BIND                                :"::"
ARROW                               :"->"

IF                                  :IF
THEN                                :THEN
ELSE                                :ELSE
IN                                  :IN
U                                   :U

ALL                                 :ALL
SOME                                :SOME
NUMBER                              :NUMBER
SUM                                 :SUM
PRODUCT                             :PRODUCT
SET                                 :SET
MAXIMUM                             :MAXIMUM
MINIMUM                             :MINIMUM
UNION                               :UNION
INTERSECTION                        :INTERSECTION
SUCH                                :SUCH{Blank}*THAT
ELSE_IF                             :ELSE{Blank}*IF

AS                                  :AS
CHOOSE                              :CHOOSE
CONCEPT                             :CONCEPT
DEFINITION                          :DEFINITION
DELAY                               :DELAY
DO                                  :DO
END                                 :END
EXCEPTION                           :EXCEPTION
EXPORT                              :EXPORT
FI                                  :FI
FOREACH                             :FOREACH
FROM                                :FROM
FUNCTION                            :FUNCTION
GENERATE                            :GENERATE
HIDE                                :HIDE
IMPORT                              :IMPORT
INHERIT                             :INHERIT
INITIALLY                           :INITIALLY
INSTANCE                            :INSTANCE
INVARIANT                           :INVARIANT
MACHINE                             :MACHINE
MESSAGE                             :MESSAGE
MODEL                               :MODEL
OD                                  :OD
OF                                  :OF
OPERATOR                            :OPERATOR
```

```
OTHERWISE                                                    :OTHERWISE
PERIOD                                                       :PERIOD
RENAME                                                       :RENAME
REPLY                                                        :REPLY
SEND                                                         :SEND
STATE                                                        :STATE
TEMPORAL                                                     :TEMPORAL
TIME                                                         :TIME
TO                                                           :TO
TRANSACTION                                                  :TRANSACTION
TRANSITION                                                   :TRANSITION
TYPE                                                         :TYPE
VALUE                                                        :VALUE
VIRTUAL                                                      :VIRTUAL
WHEN                                                         :WHEN
WHERE                                                        :WHERE

INTEGER_LITERAL                                              :{Int}
REAL_LITERAL                                                 :{Int}"."{Int}
CHAR_LITERAL                                                 :"'"."'"
STRING_LITERAL                                               :{Quote}{Char}*{Quote}

NAME                                                         :{Letter}{Alpha}*

! operator precedences
! %left means 2+3+4 is (2+3)+4.

%left                          ';', IF, DO, EXCEPTION, NAME, SEMI;
%left                          ',', COMMA;
%left                          SUCH;
%left                          IFF;
%left                          IMPLIES;
%left                          OR;
%left                          AND;
%left                          NOT;
%left                          '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%nonassoc                      IN, RANGE;
%left                          U, APPEND;
%left                          '+', '-', PLUS, MINUS;
%left                          '*', '/', MUL, DIV, MOD;
%left                          UMINUS;
%left                          EXP;
%left                          '$', '[', '(', '{', '.', DOT, WHERE;
%left                          STAR;

%%
!attribute declarations

%%
! productions of the grammar

start
            : spec
              { }
            ;
```

```
spec
            : spec module
              { }
            |
              { }
            ;
              ! A production with nothing after the "|" means the empty string
              ! is a legal replacement for the left hand side.

module
            : function
              { }
            | machine
              { }
            | type
              { }
            | definition
              { }
            | instance                                  ! of a generic module
              { }
            ;

function
            : optionally_virtual FUNCTION interface messages concepts END
              { }
            ;
              ! Virtual modules are for inheritance only, never used directly.

machine
            : optionally_virtual MACHINE interface state messages transactions temporals
concepts END
              { }
            ;

type
            : optionally_virtual TYPE interface model messages transactions temporals
concepts END
              { }
            ;

definition
            : DEFINITION interface concepts END
              { }
            ;

instance
            : INSTANCE formal_name '=' actual_name END
              { }
            | INSTANCE foreach actual_name END
              { }
            ;
              ! For making instances or partial instantiations of generic modules.
              ! The foreach clause allows defining sets of instances.

interface
            : formal_name inherits imports export
              { }
            ;
              ! This part describes the static aspects of a module's interface.
              ! The dynamic aspects of the interface are described in the messages.
              ! A module is generic iff it has parameters.
```

```
                ! The parameters can be constrained by a WHERE clause.
                ! A module can inherit the behavior of other modules.
                ! A module can import concepts from other modules.
                ! A module can export concepts for use by other modules.

inherits
              : inherits INHERIT actual_name hide renames
                { }
              |
                { }
              ;
                ! Ancestors are generalizations or simplified views of a module.
                ! A module inherits all of the behavior of its ancestors.
                ! Hiding a message or concept means it will not be inherited.
                ! Inherited components can be renamed to avoid naming conflicts.

hide
              : HIDE name_list
                { }
              |
                { }
              ;
                ! Useful for providing limited views of an actor.
                ! Different user classes may see different views of a system.
                ! Messages and concepts can be hidden.

renames
              : renames RENAME NAME AS NAME
                { }
              |
                { }
              ;
                ! Renaming is useful for preventing name conflicts when inheriting
                ! from multiple sources, and for adapting modules for new uses.
                ! The parameters, model and state components, messages, exceptions,
                ! and concepts of an actor can be renamed.

imports
              : imports IMPORT name_list FROM actual_name
                { }
              |
                { }
              ;

export
              : EXPORT name_list
                { }
              |
                { }
              ;

messages
              : messages message
                { }
              |
                { }
              ;
```

```
message
            : MESSAGE formal_message operator response
              { }
            ;

response
            : response_body
              { }
            | response_cases
              { }
            ;

response_cases
            : WHEN expression_list response_body response_cases
              { }
            | OTHERWISE response_body
              { }
            ;

response_body
            : choose reply sends transition
              { }
            ;

choose
            : CHOOSE '(' field_list restriction ')'
              { }
            |
              { }
            ;
reply
            : REPLY actual_message where
              { }
            | GENERATE actual_message where       ! used in generators
              { }
            |
              { }
            ;

sends
            : sends send
              { }
            |
              { }
            ;

send
            : optional_foreach SEND actual_message TO actual_name where
              { }
            ;

transition
            : TRANSITION expression_list              ! for describing state changes
              { }
            |
              { }
            ;
```

```
formal_message
            : optional_exception optional_formal_name formal_arguments
              { }
            ;

actual_message
            : optional_exception optional_actual_name formal_arguments
              { }
            ;

where
            : WHERE expression_list
              { }
            |                                                %prec SEMI  ! must have a lower
precedence than WHERE
              { }
            ;

optionally_virtual
            : VIRTUAL
              { }
            |
              { }
            ;

optional_exception
            : EXCEPTION
              { }
            |                                                %prec SEMI
              { }
            ;

operator
            : OPERATOR operator_list
              { }
            |
              { }
            ;

optional_foreach
            : foreach
              { }
            |
              { }
            ;

foreach
            : FOREACH '(' field_list restriction ')'
              { }
            ;
              ! foreach is used to describe a set of messages or instances

concepts
            : concepts concept
              { }
            |
              { }
            ;

concept
            : CONCEPT formal_name ':' type_spec where
```

```
              ! constants
              { }
            | CONCEPT formal_name formal_arguments where VALUE formal_arguments where
              ! functions, defined with preconditions and postconditions
              { }
            ;

model               ! data types have conceptual models for values
            : MODEL formal_arguments invariant
              { }
            ;

state       ! machines have conceptual models for states
            : STATE formal_arguments invariant initially
              { }
            ;

invariant   ! invariants are true for all states or instances
            : INVARIANT expression_list
              { }
            ;

initially   ! initial conditions are true only at the beginning
            : INITIALLY expression_list
              { }
            ;

transactions
            : transactions transaction
              { }
            |
              { }
            ;

transaction
            : TRANSACTION formal_name '=' action_list where
              { }
            ;
              ! Transactions are atomic.
              ! The where clause can specify timing constraints.

action_list
            : action_list ';' action    %prec SEMI    ! sequence
              { }
            | action
              { }
            ;

action
            : action action    %prec STAR    ! unordered set of actions
              { }
            | IF alternatives FI    ! choice
              { }
            | DO alternatives OD    ! repeated choice
              { }
            | actual_name    ! a normal message or subtransaction
              { }
            | EXCEPTION actual_name    ! an exception message
              { }
            ;
```

```
alternatives
            : alternatives OR guard action_list
              { }
            | guard action_list
              { }
            ;

guard
            : WHEN expression ARROW
              { }
            |
              { }
            ;

temporals
            : temporals temporal
              { }
            |
              { }
            ;

temporal
            : TEMPORAL NAME where response
              { }
            ;
              ! Temporal events are trigged at absolute times,
              ! in terms of the local clock of the actor.
              ! The "where" describes the triggering conditions
              ! in terms of TIME, PERIOD, and DELAY.

optional_formal_name
            : formal_name
              { }
            |
              { }
            ;

formal_name
            : NAME formal_parameters
              { }
            ;

formal_parameters     ! parameter values are determined at specification time
            : '{' field_list '}' where
              { }
            |
              { }
            ;

formal_arguments      ! arguments are evaluated at run-time
            : '(' field_list ')'
              { }
            |
              { }
            ;
```

```
field_list
            : field_list ',' field
              { }
            | field
              { }
            ;

field
            : name_list ':' type_spec
              { }
            | '$' NAME ':' type_spec
              { }
            | '?'
              { }
            ;

type_spec
            : actual_name                                   ! name of a data type
              { }
            | '?'
              { }
            ;

name_list
            : name_list NAME
              { }
            | NAME
              { }
            ;

optional_actual_name
            : actual_name
              { }
            |
              { }
            ;

actual_name
            : NAME actual_parameters
              { }
            ;

actual_parameters     ! parameter values are determined at specification time
            : '{' arg_list '}'
              { }
            |                                               %prec SEMI  ! must have a lower
precedence than '{'
              { }
            ;

actual_arguments      ! arguments are evaluated at run-time
            : '(' arg_list ')'
              { }
            |                                               %prec SEMI  ! must have a lower
precedence than '('
              { }
            ;
```

```
arg_list
            : arg_list ',' arg                          %prec COMMA
              { }
            | arg
              { }
            ;

arg
            : expression
              { }
            | pair
              { }
            ;

expression_list
            : expression_list ',' expression            %prec COMMA
              { }
            | expression                                %prec COMMA
              { }
            ;

expression
            : quantifier '(' field_list restriction BIND expression ')'
              { }
            | actual_name actual_arguments
              { }
            | actual_name '@' actual_name actual_arguments
              { }
            | NOT expression                            %prec NOT
              { }
            | expression AND expression                 %prec AND
              { }
            | expression OR expression                  %prec OR
              { }
            | expression IMPLIES expression             %prec IMPLIES
              { }
            | expression IFF expression                 %prec IFF
              { }
            | expression '<' expression                 %prec LE
              { }
            | expression '>' expression                 %prec LE
              { }
            | expression '=' expression                 %prec LE
              { }
            | expression LE expression                  %prec LE
              { }
            | expression GE expression                  %prec LE
              { }
            | expression NE expression                  %prec LE
              { }
            | expression NLT expression                 %prec LE
              { }
            | expression NGT expression                 %prec LE
              { }
            | expression NLE expression                 %prec LE
              { }
            | expression NGE expression                 %prec LE
              { }
            | expression EQV expression                 %prec LE
              { }
```

```
| expression NEQV expression                %prec LE
  { }
| '-' expression                            %prec UMINUS
  { }
| expression '+' expression                 %prec PLUS
  { }
| expression '-' expression                 %prec MINUS
  { }
| expression '*' expression                 %prec MUL
  { }
| expression '/' expression                 %prec DIV
  { }
| expression MOD expression                 %prec MOD
  { }
| expression EXP expression                 %prec EXP
  { }
| expression U expression                   %prec U
  { }
| expression APPEND expression              %prec APPEND
  { }
| expression IN expression                  %prec IN
  { }
| '*' expression                            %prec STAR
  ! *x is the value of x in the previous state
  { }
| '$' expression                            %prec DOT
  ! $x represents a collection of items rather than just one
  ! s1 = {x, $s2} means s1 = union({x}, s2)
  ! s1 = [x, $s2] means s1 = append([x], s2)
  { }
| expression RANGE expression               %prec RANGE
  ! x in [a .. b] iff x in {a .. b} iff a <= x <= b
  ! [a .. b] is sorted in increasing order
  { }
| expression '.' NAME                       %prec DOT
  { }
| expression '[' expression ']'             %prec DOT
  { }
| '(' expression ')'
  { }
| '(' expression NAME ')'   ! expression with units of measurement
   ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS
! MINUTES HOURS DAYS WEEKS
  { }
| TIME      ! The current local time, used in temporal events
  { }
| DELAY     ! The time between the triggering event and the response
  { }
| PERIOD    ! The time between successive events of this type
  { }
| literal
  { }
| literal '@' actual_name                   ! literal with explicit type
  { }
| '?'       ! An undefined value to be specified later
  { }
| '!'       ! An undefined and illegal value
  { }
| IF expression THEN expression middle_cases ELSE expression FI
  { }
;
```

```
middle_cases
            : middle_cases ELSE_IF expression THEN expression
              { }
            |
              { }
            ;

quantifier
            : ALL
              { }
            | SOME
              { }
            | NUMBER
              { }
            | SUM
              { }
            | PRODUCT
              { }
            | SET
              { }
            | MAXIMUM
              { }
            | MINIMUM
              { }
            | UNION
              { }
            | INTERSECTION
              { }
            ;

restriction
            : SUCH expression
              { }
            |
              { }
            ;

literal
            : INTEGER_LITERAL
              { }
            | REAL_LITERAL
              { }
            | CHAR_LITERAL
              { }
            | STRING_LITERAL
              { }
            | '#' NAME                                    ! enumeration type literal
              { }
            | '[' expressions ']'                         ! sequence literal
              { }
            | '{' expressions '}'                         ! set literal
              { }
            | '{' expressions ';' expression '}'        ! map literal
              { }
            | '[' pair_list ']'                                  ! tuple literal
              { }
            | '{' pair '}'                                       ! one_of literal
              { }
            ;
              ! relation literals are sets of tuples
```

```
expressions
            : expression_list
              { }
            |
              { }
            ;

pair_list
            : pair_list ',' pair
              { }
            | NAME pair
              { }
            | pair
              { }
            ;

pair
            : NAME BIND expression
              { }
            ;

operator_list
            : operator_list operator_symbol
              { }
            | operator_symbol
              { }
            ;

operator_symbol
            : NOT
              { }
            | AND
              { }
            | OR
              { }
            | IMPLIES
              { }
            | IFF
              { }
            | '<'
              { }
            | '>'
              { }
            | '='
              { }
            | LE
              { }
            | GE
              { }
            | NE
              { }
            | NLT
              { }
            | NGT
              { }
            | NLE
              { }
            | NGE
              { }
```

```
| EQV
  { }
| NEQV
  { }
| '+'
  { }
| '-'
  { }
| '*'
  { }
| '/'
  { }
| MOD
  { }
| EXP
  { }
| U
  { }
| APPEND
  { }
| IN
  { }
| RANGE
  { }
| '.'
  { }
| '['
  { }
;
```

APPENDIX B - CODE.

This Appendix contains all of the code which was written or modified to implement the type checker. There are eight actual files contained in this Appendix. Each file has a unique purpose and the specifics of their use is detailed in the first file--makefile.

1. MAKEFILE.

```
spec: macros.m4 mylibcat.c spec.m4
        /n/suns2/usr/suns2/merge/BIN/kscript -DAGLEXDEBUG -DAGYACCDEBUG \
             -t '%p 5000' -t '%a 5000' -t '%o 5000' -t '%e 5000' -s -x -z -k -e\
             -g -v -d /n/suns2/usr/suns2/merge/BIN \
             -DUSERLIB=\"/n/suns2/usr/suns2/merge/kopas/thesis_imp/mylibcat.c\" spec.m4


macros.m4: lib/head.m4 lib/attrib_psg.m4 myconst.m4 mymac.m4 lib/tail.m4
        cat lib/head.m4 lib/attrib_psg.m4 myconst.m4 mymac.m4 lib/tail.m4 >macros.m4
        chmod +r macros.m4

mylibcat.c: /n/suns2/usr/suns2/merge/BIN/locallib.c mylib.c
        cat /n/suns2/usr/suns2/merge/BIN/locallib.c mylib.c > mylibcat.c
        chmod +r mylibcat.c

output: mylib.c lib/attrib_psg.m4 myconst.m4 mymac.m4 spec.m4
        print mylib.c myconst.m4 mymac.m4 spec.m4
        cat lib/attrib_psg.m4 >! attrib_psg.m4
        print attrib_psg.m4
        rm attrib_psg.m4
```

2. ATTRIB_PSG.M4.

```
/*
 *   Macros for passing stuff around.
 */
define(passup_1,          `$$'.$2_s = $1.$2_s)

define(passup_2,          passup_1($1, $2);
                passup_1($1, $3))

define(passup_3,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4))

define(passup_4,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4);
                passup_1($1, $5))
```

```
define(passup_5,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4);
                passup_1($1, $5);
                passup_1($1, $6))

define(passup_6,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4);
                passup_1($1, $5);
                passup_1($1, $6);
                passup_1($1, $7))

define(passup_7,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4);
                passup_1($1, $5);
                passup_1($1, $6);
                passup_1($1, $7);
                passup_1($1, $8))

define(passup_8,          passup_1($1, $2);
                passup_1($1, $3);
                passup_1($1, $4);
                passup_1($1, $5);
                passup_1($1, $6);
                passup_1($1, $7);
                passup_1($1, $8);
                passup_1($1, $9))


define(passdn_1,          $1.$2_i = '$$'.$2_i)

define(passdn_2,          passdn_1($1, $2);
                passdn_1($1, $3))

define(passdn_3,          passdn_1($1, $2);
                passdn_1($1, $3);
                passdn_1($1, $4))

define(passdn_4,          passdn_1($1, $2);
                passdn_1($1, $3);
                passdn_1($1, $4);
                passdn_1($1, $5))

define(passdn_5,          passdn_1($1, $2);
                passdn_1($1, $3);
                passdn_1($1, $4);
                passdn_1($1, $5);
                passdn_1($1, $6))

define(passdn_6,          passdn_1($1, $2);
                passdn_1($1, $3);
                passdn_1($1, $4);
                passdn_1($1, $5);
                passdn_1($1, $6);
                passdn_1($1, $7))

define(passdn2_1,        $1.$3_i = '$$'.$3_i;
                $2.$3_i = '$$'.$3_i)
```

```
define(passdn2_2,         passdn2_1($1, $2, $3);
                passdn2_1($1, $2, $4))

define(passdn2_3,         passdn2_1($1, $2, $3);
                passdn2_1($1, $2, $4);
                passdn2_1($1, $2, $5))

define(passdn2_4,         passdn2_1($1, $2, $3);
                passdn2_1($1, $2, $4);
                passdn2_1($1, $2, $5);
                passdn2_1($1, $2, $6))

define(passdn2_5,         passdn2_1($1, $2, $3);
                passdn2_1($1, $2, $4);
                passdn2_1($1, $2, $5);
                passdn2_1($1, $2, $6);
                passdn2_1($1, $2, $7))

define(passdn2_6,         passdn2_1($1, $2, $3);
                passdn2_1($1, $2, $4);
                passdn2_1($1, $2, $5);
                passdn2_1($1, $2, $6);
                passdn2_1($1, $2, $7);
                passdn2_1($1, $2, $8))


define(passdn3_1,        $1.$4_i = '$$'.$4_i;
                $2.$4_i = '$$'.$4_i;
                $3.$4_i = '$$'.$4_i)

define(passdn3_2,         passdn3_1($1, $2, $3, $4);
                passdn3_1($1, $2, $3, $5))

define(passdn3_3,         passdn3_1($1, $2, $3, $4);
                passdn3_1($1, $2, $3, $5);
                passdn3_1($1, $2, $3, $6))

define(passdn3_4,         passdn3_1($1, $2, $3, $4);
                passdn3_1($1, $2, $3, $5);
                passdn3_1($1, $2, $3, $6);
                passdn3_1($1, $2, $3, $7))

define(passdn3_5,         passdn3_1($1, $2, $3, $4);
                passdn3_1($1, $2, $3, $5);
                passdn3_1($1, $2, $3, $6);
                passdn3_1($1, $2, $3, $7);
                passdn3_1($1, $2, $3, $8))

define(passdn4_1,         passdn2_1($1, $2, $5);
                          passdn2_1($3, $4, $5))
define(passdn4_2,         passdn4_1($1, $2, $3, $4, $5);
                passdn4_1($1, $2, $3, $4, $6))
define(passdn5_1,         passdn3_1($1, $2, $3, $6);
                passdn2_1($4, $5, $6))


/*
 *   Passovr is used for passing attributes from one non-terminal to another.
 * The order is (from,to,attribute,...)
 */
define(passovr_1,        $2.$3_i = $1.$3_s)
```

```
define(passovr_2,       passovr_1($1, $2, $3);
                passovr_1($1,$2,$4))
define(passovr_3,       passovr_1($1, $2, $3);
                passovr_1($1, $2, $4);
                passovr_1($1, $2, $5))
define(passovr_4,       passovr_1($1, $2, $3);
                passovr_1($1, $2, $4);
                passovr_1($1, $2, $5);
                passovr_1($1, $2, $6))
define(passovr_5,       passovr_1($1, $2, $3);
                passovr_1($1, $2, $4);
                passovr_1($1, $2, $5);
                passovr_1($1, $2, $6);
                passovr_1($1, $2, $7))
define(passovr_6,       passovr_1($1, $2, $3);
                passovr_1($1, $2, $4);
                passovr_1($1, $2, $5);
                passovr_1($1, $2, $6);
                passovr_1($1, $2, $7);
                passovr_1($1, $2, $8))


/*
 *   Pass information about in pre-order.
 * Parent is first argument, then children left to right.
 * Attribute comes last.  Macro appends _i and _s to attribute
 * names as appropriate...
 */

define(passio0_1,          `$$'.$1_s = `$$'.$1_i)

define(passio1_1,          $1.$2_i = `$$'.$2_i;
                `$$'.$2_s = $1.$2_s)

define(passio2_1,          $1.$3_i = `$$'.$3_i;
                $2.$3_i = $1.$3_s;
                `$$'.$3_s = $2.$3_s)

define(passio3_1,          $1.$4_i = `$$'.$4_i;
                $2.$4_i = $1.$4_s;
                $3.$4_i = $2.$4_s;
                `$$'.$4_s = $3.$4_s)


define(passio0_2,       passio0_1($1);
                passio0_1($2))

define(passio0_3,       passio0_1($1);
                passio0_1($2);
                passio0_1($3))

define(passio0_4,       passio0_1($1);
                passio0_1($2);
                passio0_1($3);
                passio0_1($4))
```

```
define(passio0_6,       passio0_1($1);
                passio0_1($2);
                passio0_1($3);
                passio0_1($4);
                passio0_1($5);
                passio0_1($6))


define(passio1_2,       passio1_1($1, $2);
                passio1_1($1, $3))

define(passio1_3,       passio1_1($1, $2);
                passio1_1($1, $3);
                passio1_1($1, $4))

define(passio1_4,       passio1_1($1, $2);
                passio1_1($1, $3);
                passio1_1($1, $4);
                passio1_1($1, $5))

define(passio1_5,       passio1_1($1, $2);
                passio1_1($1, $3);
                passio1_1($1, $4);
                passio1_1($1, $5);
                passio1_1($1, $6))


define(passio2_2,       passio2_1($1, $2, $3);
                passio2_1($1, $2, $4))

define(passio2_3,       passio2_1($1, $2, $3);
                passio2_1($1, $2, $4);
                passio2_1($1, $2, $5))

define(passio2_4,       passio2_1($1, $2, $3);
                passio2_1($1, $2, $4);
                passio2_1($1, $2, $5);
                passio2_1($1, $2, $6))

define(passio2_5,       passio2_1($1, $2, $3);
                passio2_1($1, $2, $4);
                passio2_1($1, $2, $5);
                passio2_1($1, $2, $6);
                passio2_1($1, $2, $7))


define(passio3_2,       passio3_1($1, $2, $3, $4);
                passio3_1($1, $2, $3, $5))

define(passio3_3,       passio3_1($1, $2, $3, $4);
                passio3_1($1, $2, $3, $5);
                passio3_1($1, $2, $3, $6))

define(passio3_4,       passio3_1($1, $2, $3, $4);
                passio3_1($1, $2, $3, $5);
                passio3_1($1, $2, $3, $6);
                passio3_1($1, $2, $3, $7))
```

```
define(passio3_5,       passio3_1($1, $2, $3, $4);
                passio3_1($1, $2, $3, $5);
                passio3_1($1, $2, $3, $6);
                passio3_1($1, $2, $3, $7);
                passio3_1($1, $2, $3, $8))



/* pass up strings, concatenated together */
define(catstrup2_1,    '$$'.$3_s = $1.$3_s ^ $2.$3_s)
define(catstrup3_1,    '$$'.$4_s = $1.$4_s ^ $2.$4_s ^ $3.$4_s)


/* pass up maps, concatenated together */
define(catmapup2_1,      '$$'.$3_s = $1.$3_s +| $2.$3_s)
define(catmapup3_1,      '$$'.$4_s = $1.$4_s +| $2.$4_s +| $3.$4_s)
define(passovr2x_1, passovr_1($1,$2,$4);
                    passovr_1($1,$3,$4))
define(passovr3x_1, passovr2x_1($1,$2,$3,$5);
                    passovr_1($1,$4,$5))
define(passovr4x_1, passovr2x_1($1,$2,$3,$6);
                    passovr2x_1($1,$4,$5,$6))
define(passovr2x_2, passovr_2($1,$2,$4,$5);
                   passovr_2($1,$3,$4,$5))
define(passovr3x_2, passovr3x_1($1, $2, $3, $4, $5);
                    passovr3x_1($1, $2, $3, $4, $6))
define(passovr4x_2, passovr4x_1($1, $2, $3, $4, $5, $6);
                    passovr4x_1($1, $2, $3, $4, $5, $7))



/*
 * weave -- a partial version of passio.
 *          weave assumes the first nonterminal generates the attribute,
 *          and all non-terminals listed use the product of the previou
 *          nonterminal, but the attribute is not returned to the parent
 *          nonterminal after it's use by the last nonterminal.
 */
define(weave3_1, passovr($1,$2,$4);
                 passovr($2,$3,$4))

define(weave4_1, weave3_1($1, $2, $3, $5);
                 passovr($3, $4, $5))

define(weave3_2, weave3_1($1, $2, $3, $4);
                 weave3_1($1, $2, $3, $5))

define(weave4_2, weave4_1($1, $2, $3, $4, $5);
                 weave4_1($1, $2, $3, $4, $6))

define(passio4_1, $1.$5_i = '$$'.$5_i;
                  $2.$5_i = $1.$5_s;
                  $3.$5_i = $2.$5_s;
                  $4.$5_i = $3.$5_s;
                  '$$'.$5_s = $4.$5_s)
define(passio4_2, passio4_1($1, $2, $3, $4, $5);
                  passio4_1($1, $2, $3, $4, $6))
```

```
define(passio5_1, $1.$6_i = '$$'.$6_i;
                  $2.$6_i = $1.$6_s;
                  $3.$6_i = $2.$6_s;
                  $4.$6_i = $3.$6_s;
                  $5.$6_i = $4.$6_s;
                  '$$'.$6_s = $5.$6_s)
define(passio5_2, passio5_1($1, $2, $3, $4, $5, $6);
                  passio5_1($1, $2, $3, $4, $5, $7))
define(passio6_1, $1.$7_i = '$$'.$7_i;
                  $2.$7_i = $1.$7_s;
                  $3.$7_i = $2.$7_s;
                  $4.$7_i = $3.$7_s;
                  $5.$7_i = $4.$7_s;
                  $6.$7_i = $5.$7_s;
                  '$$'.$7_s = $6.$7_s)
define(passio6_2, passio6_1($1, $2, $3, $4, $5, $6, $7);
                  passio6_1($1, $2, $3, $4, $5, $6, $8))
```

3. MYMAC.M4.

```
/* Macros used for making declarations shorter.
 */
define(IP_STBL_INFO,     ip_stbl_class_s : int->int;
    ip_stbl_names_s  : int->string;
    ip_stbl_params_s : int->string;
    ip_stbl_result_s : int->string;
    ip_stbl_class_i  : int->int;
    ip_stbl_names_i  : int->string;
    ip_stbl_params_i : int->string;
    ip_stbl_result_i : int->string)

define(STBL_INFO,  stbl_i : string->string->string;
    stbl_class_i  : int->int;
    stbl_names_i  : int->string;
    stbl_params_i : int->string;
    stbl_result_i : int->string)

define(VISIBILITY_TBLS,     visible_types_i : string->string;
    visible_types_s : string->string;
    visible_names_i : string->string;
    visible_names_s : string->string)

define(IP_MCMXREF_TBLS,     ip_mcmxref_s : string->string;
       ip_mcmxref_i : string->string)




/*
*  Macro's used for various tasks including defining names, etc.
*
*
* mk_simple & mk_complex are very much alike.  mk_complex has arguments ($5)
*  though.
*/
```

```
define(mk_simple_decl_io, $1_s = ($2 == NULL_STRING)
                        -> ($1_i ($3) == NULL_STRING)
                           -> {($3 : [XREF_DELIMITER, i2s($4)])} +| $1_i
                           '#'{($3 : [XREF_DELIMITER, i2s($4),
                                      PATTERN_DELIMITER, $1_i ($3)])} +| $1_i
                        '#' $1_i)

define(mk_simple_decl, $5 = ($2 == NULL_STRING)
                        -> ($1 ($3) == NULL_STRING)
                           -> {($3 : [XREF_DELIMITER, i2s($4)])} +| $1
                           '#'{($3 : [XREF_DELIMITER, i2s($4),
                                      PATTERN_DELIMITER, $1 ($3)])} +| $1
                        '#' $1)

define(mk_complex_decl, $1_s  = ($2 == NULL_STRING)
                        -> ($1_i ($3) == NULL_STRING)
                           -> {($3 : [$5, XREF_DELIMITER, i2s($4)])} +| $1_i
                           '#'{($3 : [$1_i ($3), PATTERN_DELIMITER, $5,
                                      XREF_DELIMITER, i2s($4)])} +| $1_i
                        '#' $1_i)

define(add_elem, $4 = $1 +| {($2 : $3)} )


/*
 * Macro's used for passing around the symbol table information.
 * This is the stuff that never changes, e.g. is not modified during progress
 * of type checking a module.
 *   Symbol Table Information Consists of
 *      1. stbl
 *      2. stbl_names
 *      3. stbl_result
 *      4. stbl_class
 *      5. stbl_params
 */


/* symbol table building macros */

define(stbl_build0, passio0_4(ip_stbl_class,ip_stbl_names,ip_stbl_params,
                              ip_stbl_result))
define(stbl_build1, passio1_4($1,ip_stbl_class,ip_stbl_names,ip_stbl_params,
                              ip_stbl_result))
define(stbl_build2, passio2_4($1,$2,ip_stbl_class,ip_stbl_names,
                              ip_stbl_params, ip_stbl_result))
define(stbl_build3, passio3_4($1,$2,$3,ip_stbl_class,ip_stbl_names,
                              ip_stbl_params, ip_stbl_result))
define(stbl_build4, passio4_2($1,$2,$3,$4,ip_stbl_class,ip_stbl_names);
                    passio4_2($1,$2,$3,$4,ip_stbl_params, ip_stbl_result))
define(stbl_build5, passio5_2($1,$2,$3,$4,$5,ip_stbl_class,ip_stbl_names);
                    passio5_2($1,$2,$3,$4,$5,ip_stbl_params, ip_stbl_result))
define(stbl_build6, passio6_2($1,$2,$3,$4,$5,$6,ip_stbl_class,ip_stbl_names);
                    passio6_2($1,$2,$3,$4,$5,$6,ip_stbl_params, ip_stbl_result))

define(passdn_stbl1, passdn_5($1, stbl, stbl_result, stbl_class, stbl_names,
                              stbl_params))
define(passdn_stbl2, passdn2_5($1, $2, stbl, stbl_result, stbl_class, stbl_names,
                              stbl_params))
define(passdn_stbl3, passdn3_5($1, $2, $3, stbl, stbl_result, stbl_class,
                              stbl_names,stbl_params))
```

```
define(passdn_stbl4, passdn3_5($1, $2, $3, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params);
                     passdn_5($4, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params))
define(passdn_stbl5, passdn3_5($1, $2, $3, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params);
                     passdn2_5($4, $5, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params))
define(passdn_stbl6, passdn3_5($1, $2, $3, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params);
                     passdn3_5($4, $5, $6, stbl, stbl_result, stbl_class,
                               stbl_names,stbl_params))
```

4. MYCONST.M4.

```
/* Symbolic Constants used in Program -- Are always capitalized. */
define(NULL_STRING, `""')
define(UNDEFINED_TYPE, `"Type Name Undefined."')
define(SPEC_LIBRARY_MODULE_type, `"type"')
define(GLOBAL_TYPE_NAMES, `"#global#"')
define(CURRENT_MODULE_TAG, `"#current_module#"')
define(FALSE, 0)
define(FUNCTION_CLASS, 1)
define(MACHINE_CLASS, 2)
define(TYPE_CLASS, 3)
define(DEFINITION_CLASS, 4)
define(INSTANCE_CLASS, 5)
define(MESSAGE_CLASS, 6)
define(CONCEPT_CLASS1, 7)
define(CONCEPT_CLASS2, 8)
define(VARIABLE_CLASS, 9)
define(TRANSACTION_CLASS, 10)
define(TEMPORAL_CLASS, 11)
define(PATTERN_DELIMITER, `"*"')
define(XREF_DELIMITER, `"-"')
define(ELEM_DELIMITER, `";"')
define(REF_SYMBOL, `"^"')
define(ACTUAL_DELIM, `";"')
define(PAIR_DELIM, `";"')


/*
 * Errors & Warning Messages Generated by Kodiyak code.
 */
define(UNRESOLVED_TYPE, -1)
define(UNDECLARED_TYPE, 3)
```

5. HEAD.M4.

```
divert(-1)
/*
 *
 *  Copyright 1986, Robert Herndon
 *         (C) 1986, Robert Herndon
 *  Modified by Robert Kopas 1989.
 *       Purpose - To allow Consistency and implement macros needed
 *                 for the type checker.
 */
```

6. TAIL.M4.

```
/*
 *   Many of m4's keywords are commonly used words.  Remove
 * all builtin macro names to suppress any unexpected side-effects.
 */
undefine('#')
undefine('changequote')
undefine('define')
undefine('divnum')
undefine('dnl')
undefine('dumpdef')
undefine('errprint')
undefine('eval')
undefine('ifdef')
undefine('ifelse')
undefine('include')
undefine('incr')
undefine('index')
undefine('len')
undefine('maketemp')
undefine('sinclude')
undefine('substr')
undefine('syscmd')
undefine('translit')
undefine('undivert')
divert(0)
undefine('divert')
undefine('undefine')
```

7. MYLIB.C.

```
/* Symbolic Constant Declarations -- It is extremely important that
   these concur with those defined in mymac.m4
*/
#define MYCHARLENMAX        10000
#define MAX_XREF_NUM_LEN    20

#define PATTERN_DELIMITER   '*'
#define XREF_DELIMITER      '-'
#define ELEM_DELIMITER      ';'
#define END_ELEMENT(s)      (*s == ELEM_DELIMITER)
#define END_ACTUALS(s)      (*s == '\0')
#define END_FORMALS(s)      (*s == XREF_DELIMITER)
#define END_PATTERN(s)      (*s == PATTERN_DELIMITER)
```

```
#define UNDEFINED_TYPE      "Type Name Undefined."
#define CURRENT_MODULE_TAG  "#current_module#"
#define MESSAGE_CLASS       6

/* errors -- those that are enumerated in mymac.m4 must concur with these
   also.
*/
#define NAME_REDEFINED      1
#define CNAME_REDEFINED     2
#define UNDECLARED_TYPE     3
#define NONSPECIFIC_REFERENCE 4

/*
 * Warning Messages
 */
#define UNRESOLVED_TYPE    -1



int last_xref = 0;

/*
 * v_get_new_xref has an argument named unused just to
 * satisfy Kodiyak syntactic requirements.
 * Everytime the routine is called, it will obtain a new, unique xref.
 */
int vget_new_xref( unused )
char *unused;
{
    return (++last_xref);
}

char mychars[MYCHARLENMAX];
int mycharlen = 0;


wxrefs_dump (xreftostrmap)
xobject xreftostrmap;
{
   int cur_index;
   xstring lookup_elem;

   for (cur_index = 1; cur_index <= last_xref; cur_index++) {
       lookup_elem = ximapslkup(xreftostrmap, cur_index);
       printf("\t%d : ",cur_index);
       woutput(lookup_elem);
       printf("\n");
       fflush(stdout);
   }
}

wxrefi_dump (xreftointmap)
xobject xreftointmap;
{
   int cur_index;
   int lookup_elem;

   for (cur_index = 1; cur_index <= last_xref; cur_index++) {
       lookup_elem = ximapilkup(xreftointmap, cur_index);
       printf("\t%d : %d\n", cur_index, lookup_elem);
       fflush(stdout);
```

```
      }
}

wsmapi_dump( mapname)
xobject mapname;
{
   struct xmflatten f;
   xobject p;
   int defval;
   int defined;
   int range_val;

   if (!XHEAP(mapname) && !XPAIR(mapname) )  {
      xerr ("smapi_dump -- Non Map Argument \n", 0, 0, 0);
   }
   defined = XFALSE;
   defval = 0;
   xminit (&f, mapname);
   for (p = xmnext(&f); p.op_type; p = xmnext(&f) ) {
      if (!XPAIR(p))
         xerr("smapi_dump -- Corrupt map. \n",0,0,0);
      if (p.op_type[0].op_type) {
         range_val = *(p.op_type[1].ip_type);
         printf("\t %s   ->   %d\n", xtstrflatten(p.op_type[0]),range_val);
         fflush (stdout);
         }
      else if (!defined) {
         defval = *(p.op_type[1].ip_type);
         defined = XTRUE;
         }
      }  /* end for loop */
   printf("\t DEFAULT  ->   %d\n", defval);
   fflush(stdout);
}

wsmaps_dump( mapname)
xobject mapname;
{
   struct xmflatten f;
   xobject p;
   xobject defval;
   int defined;
   xheap range_val;

   if (!XHEAP(mapname) && !XPAIR(mapname) )  {
      xerr ("smapi_dump -- Non Map Argument \n", 0, 0, 0);
   }
   defined = XFALSE;
   defval.op_type = (xheap) 0;
   xminit (&f, mapname);
   for (p = xmnext(&f); p.op_type; p = xmnext(&f) ) {
      if (!XPAIR(p))
         xerr("smaps_dump -- Corrupt map. \n",0,0,0);
      if (p.op_type[0].op_type) {
         range_val = p.op_type[1].op_type;
         printf("\t %s   ->   ", xtstrflatten(p.op_type[0]));
         xheapprint(stdout, range_val);
         printf("\n");
         fflush (stdout);
         }
      else if (!defined) {
```

```
            defval = p.op_type[1];
            defined = XTRUE;
            }
        }  /* end for loop */
    printf("\tDEFAULT  ->    ");
    xheapprint(stdout, defval);
    printf("\n");
    fflush(stdout);
}


/*
 * copy_pattern -- designed to copy a specific pattern to a destination
 *       address.  It returns the length of the pattern copied or 0.
 *       if the pattern is a null string (or equivalent), a '\0' is copied.
 */
int copy_pattern (d_addr, s_addr, max_len)
char *d_addr, *s_addr;
int  max_len;
{
    int char_cnt = 0;

    for (;((*s_addr != PATTERN_DELIMITER) && (*s_addr != '\0') && (max_len > 0))
         ; s_addr++, d_addr++, max_len--, char_cnt++)
        *d_addr = *s_addr;
    if (max_len > 0) {
       *d_addr = '\0';
       return (char_cnt + 1);  /* include '\0' */
    }
    return (max_len);
}



char *verror_message(err_num, line_no, xa, xb, xc, xd  )
int err_num;
int line_no;
xobject xa, xb, xc, xd;
{

    switch (err_num) {
        case NAME_REDEFINED:
            return vfmt("*** ERROR ***  Line %d : Name Already Defined -- %s
                        (by item %s)\n",
                        line_no, xa, xb);
            break;
        case CNAME_REDEFINED:
            return vfmt("*** ERROR ***  Line %d : Name Already Defined -- %s(%s)
                         (by item %s)\n",
                        line_no, xa, xb, xc);
            break;
        case UNDECLARED_TYPE:
            return vfmt("*** ERROR ***  Line %d : Type Name Undeclared -- %s\n",
                         line_no, xa);
            break;
        case NONSPECIFIC_REFERENCE:
            return vfmt("*** ERROR *** Line %d : Do you mean %s\n", line_no, xa);
        case UNRESOLVED_TYPE:
            return vfmt("** WARNING **  Line %d : Unresolved type used\n", line_no);
            break;
        default:
```

```
            break;
    }
    return "";
}


#define more_sig_patterns(s)    (*s != '\0')
#define next_pattern(s)         for (;*s != '\0';) \
                                    if (*s++ == PATTERN_DELIMITER) \
                                        break
#define next_element(s)         for (;((*s != '\0') && (*s != XREF_DELIMITER));) \
                                    if (*s++ == ELEM_DELIMITER) \
                                        break

int num_copied = 0;
#define cp2mychars(s)           if ((num_copied = copy_pattern(&mychars[mycharlen],\
                                     s, MYCHARLENMAX - mycharlen))) \
                                    mycharlen = mycharlen + num_copied;  \
                                else \
                                    xerr("MYCHARLENMAX exceeded -- increase
MYCHARLENMAX.",\
                                         0,0,0)


/*
 * Name declaration routines.
 *   These routines check for the existence of a signature.
 *   If the signature exists, they return an error message stating that
 *      the signature cannot be redefined.  Otherwise they return ""
 */

xstring vcheck_simple_decl (name_map, name, line_no, xref2name_map)
xobject name_map, name;
int line_no;
xobject xref2name_map;
{
    char xref_number [MAX_XREF_NUM_LEN];
    char *patterns;
    char *tmp_args;

    patterns = xtstrflatten (xsmapslkup(name_map, name));
    while more_sig_patterns(patterns) {
        if (*patterns == XREF_DELIMITER) {  /* empty pattern -- no args */
            for (tmp_args = patterns; *tmp_args++ != XREF_DELIMITER; )
                ;
            if (copy_pattern(xref_number, tmp_args, MAX_XREF_NUM_LEN))
               return (xstring) verror_message(NAME_REDEFINED, line_no, name, ximapslkup(
                       xref2name_map, atoi(xref_number)));
            else
               xerr("Exceeded MAX_XREF_NUM_LEN -- Increase value...",0,0,0);
        }
        else
            next_pattern(patterns);
    }
    return (xstring) "";
}
```

```
xstring vcheck_complex_decl (name_map, name, f_args, line_no, xref2name_map)
xobject name_map, name, f_args;
int line_no;
xobject xref2name_map;
{
    char *patterns;
    char *new_sig;
    char xref_number[MAX_XREF_NUM_LEN];
    char *tmp_name, *tmp_args;
    int  retval;

    mycharlen = 0;
    new_sig = mychars;
    cp2mychars(xtstrflatten(f_args));
    patterns = xtstrflatten(xsmapslkup(name_map, name));
    while more_sig_patterns (patterns) {
        if (match_f_fp (new_sig, patterns)) {
            for (tmp_args = patterns; *tmp_args++ != XREF_DELIMITER; )
                ;
            if (copy_pattern(xref_number, tmp_args, MAX_XREF_NUM_LEN)) {
                tmp_name = &mychars[mycharlen];
                cp2mychars(xtstrflatten(name));
                /* remove xref value and delimiter from args. */
                for (tmp_args = new_sig;((*tmp_args != XREF_DELIMITER) &&
                                         (*tmp_args != '\0')) ;)
                    tmp_args++;
                *tmp_args = '\0';
                return (xstring) verror_message(CNAME_REDEFINED, line_no, name, new_sig,
                    ximapslkup(xref2name_map, atoi(xref_number)));
            }
            else
                xerr("Exceeded MAX_XREF_NUM_LEN -- Increase value...",0,0,0);
        }
        else {
            next_pattern(patterns);
        }
    } /* end while */
    return (xstring) "";
}



/* type equal is strict equality.  No allowances are made (or should be)
   for subtypes or equivalences. */
int type_equal (arg1, arg2)
char *arg1, *arg2;
{
    /* position both args at beginning of type. */
    while (*arg1++ != ':')
        ;
    while (*arg2++ != ':')
        ;

    for (; *arg1 == *arg2; arg1++, arg2++)
        if ((*arg1 == ELEM_DELIMITER) || (*arg1 == XREF_DELIMITER))
            return(1);
    /* an exception condition. */
    if ((*arg1 == '\0') && (*arg2 == XREF_DELIMITER))
        return(1);
    return(0);
```

```
}


int match_f_fp (temp_new, temp_old)
char *temp_new, *temp_old;
{
    while ((*temp_new != '\0') && (*temp_old != XREF_DELIMITER)) {
        if (type_equal(temp_new, temp_old)) {
            if ((*temp_new == '$') && (*temp_old == '$')) {
                next_element(temp_new);
                next_element(temp_old);
            }
            else if (*temp_new == '$') {
                /* a recursive analysis must be done here. */
                return (0);  /*for now */
            }
            else if (*temp_old == '$') {
                /* another recursive analysis */
                return (0); /*for now */
            }
            else {
                next_element(temp_new);
                next_element(temp_old);
            }
        }
        /* the following two cases are for 0 arguments. */
        else if (*temp_new == '$') {
            next_element(temp_new);
         }
        else if (*temp_old == '$') {
            next_element(temp_old);
         }
        else
            return(0);
    } /* end while */
    if ((*temp_new == '\0') && (*temp_old == XREF_DELIMITER))
        /* both at end of formals */
        return(1);
    return(0);
}




/* routines used in resolving types and references. */
char *myalloc (req_size)
int req_size;
{
    char *p;
    if ((p = (char *) alloc (req_size)) == NULL)
         xerr("No more Dynamic Storage Available", 0,0,0);
    return(p);
}


char *save_string (s)
char *s;
{
```

```
    char *p;

    p = myalloc(strlen(s) + 1);
    strcpy(p, s);
    return (p);
}


char *loose_string(s)
char *s;
{
    free(s);
    return(NULL);
}



/* at -- locate a character in a string */
char *at(s, seek_char)
char *s;
char seek_char;
{
    for (;*s != '\0'; s++)
        if (*s == seek_char)
            return(s);
    return(NULL);
}


/* substr -- return a substring of the original string */
char *substr (start_pos, last_pos)
char *start_pos, *last_pos;
{
    char *ret_string;
    char *tmp_ptr;

    if (start_pos == NULL)
        return (NULL);
    else if (last_pos == NULL)
        return (save_string(start_pos));
    else {
        ret_string = myalloc(last_pos - start_pos + 2);
        for (tmp_ptr = ret_string; start_pos <= last_pos ; last_pos++, tmp_ptr++)
            *tmp_ptr = *start_pos;
    }
}


char *element_substr(s)
char *s;
{
    char *last_elem;

    if ((last_elem = at(s, ELEM_DELIMITER)) != NULL)
        return(substr(s, --last_elem));
    return(substr(s, last_elem));
}
```

```
char *get_arg_type (actuals, cur_arg)
char *actuals;
int  cur_arg;
{
    int tmp;
    for (tmp = 1; ((tmp != cur_arg) && !END_ACTUALS(actuals)); tmp++)
        next_element(actuals);
    if END_ACTUALS(actuals)
        return(NULL);
    return(element_substr(actuals));
}


int is_pair(s)
char *s;
{
    for (; !END_FORMALS(s) && !END_ACTUALS(s) && !END_ELEMENT(s) ; s++)
        if ((*s == ':') && (*(s + 1) == ':'))
            return (1);
    return(0);
}


int names_match (element1, element2)
char *element1, *element2;
{
    for (; *element1 == *element2 ;)
        if (*element1 == ':')
           return(1);
    return(0);
}


int mystrcmp (arg1, arg2)
char *arg1, *arg2;
{
    for (; ((*arg1 == *arg2) || (END_FORMALS(arg1) && END_ACTUALS(arg2)
        if (END_FORMALS(arg1))
            return(1);
    return(0);
}


int pullout_xref (str)
char *str;
{
    char *start_pos, *end_pos;
    char tmp_char;
    int retval;

    start_pos = at(str, XREF_DELIMITER);
    end_pos = at(str, PATTERN_DELIMITER);
    tmp_char = *end_pos;
    *end_pos = '\0';
    retval = vs2i(start_pos);
    *end_pos = tmp_char;
    return(retval);
}
```

```
int type_match (formal, actual)
char *formal, *actual;
{
    for (; *formal++ != ':' ;)
        ;   /*move past name */
    if (mystrcmp(formal, actual))
        return(1);
    else if (mystrcmp(formal, UNDEFINED_TYPE) ||
             mystrcmp(actual, UNDEFINED_TYPE) )
        return(1);
    return(0);
}



/*
 function match_fa :
     -- checks if formals match actuals.
     -- returns: true or false.
*/
int match_fa (formals, actuals)
char *formals, *actuals;
{
    while (!END_FORMALS(formals) && !END_ACTUALS(actuals)) {
        if (*formals == '$')
            if (is_pair(actuals))
                if (names_match(formals, actuals)) {
                    next_element(formals);
                    next_element(actuals);
                }
                else
                    next_element(formals);
            else if (type_match (formals, actuals))
                next_element(actuals);
            else
                next_element(formals);
        else {
            if (is_pair(actuals)) { /* name must bind */
                if (!names_match(formals, actuals))
                    return(0);
                else {  /* advance actuals past bind */
                    for (; *actuals++ != ':' ;)
                        ;
                    actuals++;
                }
            } /* end if is_pair */
            if (types_match(formals, actuals)) {
                next_element(formals);
                next_element(actuals);
            }
            else
                return(0);
        }
```

```
    }
    if (END_FORMALS(formals) && END_ACTUALS(actuals))
        return(1);
    return(0);
}


int Analyze_patterns(pattern_string, actual_args)
char *pattern_string;
char *actual_args;
{
    char *tmp_ptr, *cur_pattern;

    if ((cur_pattern = pattern_string) == NULL)
        return(0);
    while (more_sig_patterns(cur_pattern)) {
        if (match_fa (cur_pattern, actual_args))
            return (pullout_xref(cur_pattern));
        else
            next_pattern(cur_pattern);
    }
    return(0);
}



xstring vseek_symbol (name, actual_args, visible_names, stbl, stbl_classes,
                      stbl_names, line_num)
xobject name, actual_args;
xobject visible_names;
xobject stbl, stbl_classes, stbl_names;
int     line_num;
{
    xstring patterns;
    int xref_value, cur_arg, tmp_xref_val;
    char *arg_type_name;
    char *flat_patterns, *flat_actuals;
    char *other_overloadings = NULL;

    patterns = xsmapslkup(name, visible_names);
    flat_actuals = save_string(xtstrflatten(actual_args));
    xref_value = Analyze_patterns(flat_patterns, flat_actuals);
    cur_arg = 1;

    /* search all other modules for overloadings */
    while ((arg_type_name = get_arg_type(flat_actuals, cur_arg)) != NULL) {
        if (strcmp(arg_type_name, xtstrflatten(xsmapslkup(visible_names,
                                                CURRENT_MODULE_TAG)))) {
            /* not current module */
            patterns = xsmapslkup(xsmapxlkup(stbl, arg_type_name), name);
            flat_patterns = loose_string(flat_patterns);
            flat_patterns = save_string(xtstrflatten(patterns));
            tmp_xref_val = Analyze_patterns(flat_patterns, flat_actuals);
            if (ximapilkup(stbl_classes, tmp_xref_val) == MESSAGE_CLASS) {
                if (xref_value && tmp_xref_val) {
                    /* multiple overloadings */
                    if (other_overloadings == NULL)
                        other_overloadings = vfmt("%s or %s",
                           ximapslkup(stbl_names, xref_value),
```

```
                        ximapslkup(stbl_names, tmp_xref_val} );
                else
                    other_overloadings = vfmt("%s, %s",
                        ximapslkup(stbl_names, tmp_xref_val), other_overloadings};
            }
            else if (!xref_value}
                xref_value = tmp_xref_val;
        }
    }
    cur_arg++;
} /* end of while loop */

if (other_overloadings == NULL)
    return vi2s(xref_value};
return verror_message(NONSPECIFIC_REFERENCE, line_num, other_overloadings);
}
```

8. SPEC.M4.

```
! version stamp $Header: spec.k,v 1.10 89/02/11 20:11:31 berzins Locked $
! Kopas- Revised Grammar iaw v 1.11 05 April 89
! Kopas- Completed Declarations 20 April 89

! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

!m4 inclusion files: mymac.m4
include(macros.m4)


! definitions of lexical classes

%define        Digit                                    :[0-9]
%define        Int                                      :{Digit}+
%define        Letter                                   :[a-zA-Z]
%define        Alpha                                    :({Letter}|{Digit}|"_")
%define        Blank                                    :[ \t\n]
%define        Quote                                    :["]
%define        Backslash                                :"\\"
%define        Char                                     :([^"\\]|{Backslash}{Quote}|{Backs
lash}{Backslash})

! definitions of white space and comments

                                                        :{Blank}+
                                                        :"--".*"\n"

! definitions of compound symbols and keywords
```

```
! I had to add the following terminal names to get line numbers...
LBRACK                                                  :"["
DOTMARK                                                 :"."
SLASH                                                   :"/"
STARMARK                                                :"*"
MINUSMARK                                               :"-"
PLUSMARK                                                :"+"
EQUALS                                                  :"="
GT                                                      :">"
LT                                                      :"<"
QUESTION_MARK                                           :"?"

!end of my additions

AND                                                     :"&"
OR                                                      :"|"
NOT                                                     :"~"
IMPLIES                                                 :"=>"
IFF                                                     :"<=>"

LE                                                      :"<="
GE                                                      :">="
NE                                                      :"~="
NLT                                                     :"~<"
NGT                                                     :"~>"
NLE                                                     :"~<="
NGE                                                     :"~>="
EQV                                                     :"=="
NEQV                                                    :"~=="

RANGE                                                   :".."
APPEND                                                  :"||"
MOD                                                     :{Backslash}|MOD
EXP                                                     :"**"

BIND                                                    :"::"
ARROW                                                   :"->"

IF                                                      :IF
THEN                                                    :THEN
ELSE                                                    :ELSE
IN                                                      :IN
U                                                       :U

ALL                                                     :ALL
SOME                                                    :SOME
NUMBER                                                  :NUMBER
SUM                                                     :SUM
PRODUCT                                                 :PRODUCT
SET                                                     :SET
MAXIMUM                                                 :MAXIMUM
MINIMUM                                                 :MINIMUM
UNION                                                   :UNION
INTERSECTION                                            :INTERSECTION
SUCH                                                    :SUCH{Blank}*THAT
ELSE_IF                                                 :ELSE{Blank}*IF

AS                                                      :AS
CHOOSE                                                  :CHOOSE
CONCEPT                                                 :CONCEPT
DEFINITION                                              :DEFINITION
```

```
DELAY                                                   :DELAY
DO                                                      :DO
END                                                     :END
EXCEPTION                                               :EXCEPTION
EXPORT                                                  :EXPORT
FI                                                      :FI
FOREACH                                                 :FOREACH
FROM                                                    :FROM
FUNCTION                                                :FUNCTION
GENERATE                                                :GENERATE
HIDE                                                    :HIDE
IMPORT                                                  :IMPORT
INHERIT                                                 :INHERIT
INITIALLY                                               :INITIALLY
INSTANCE                                                :INSTANCE
INVARIANT                                               :INVARIANT
MACHINE                                                 :MACHINE
MESSAGE                                                 :MESSAGE
MODEL                                                   :MODEL
OD                                                      :OD
OF                                                      :OF
OPERATOR                                                :OPERATOR
OTHERWISE                                               :OTHERWISE
PERIOD                                                  :PERIOD
RENAME                                                  :RENAME
REPLY                                                   :REPLY
SEND                                                    :SEND
STATE                                                   :STATE
TEMPORAL                                                :TEMPORAL
TIME                                                    :TIME
TO                                                      :TO
TRANSACTION                                             :TRANSACTION
TRANSITION                                              :TRANSITION
TYPE                                                    :TYPE
VALUE                                                   :VALUE
VIRTUAL                                                 :VIRTUAL
WHEN                                                    :WHEN
WHERE                                                   :WHERE

INTEGER_LITERAL                                         :{Int}
REAL_LITERAL                                            :{Int}"."{Int}
CHAR_LITERAL                                            :"'"."'"
STRING_LITERAL                                          :{Quote}{Char}*{Quote}

NAME                                                    :{Letter}{Alpha}*

! operator precedences
! %left means 2+3+4 is (2+3)+4.

%left           ';', IF, DO, EXCEPTION, NAME, SEMI;
%left           ',', COMMA;
%left           SUCH;
%left           IFF;
%left           IMPLIES;
%left           OR;
%left           AND;
%left           NOT;
%left           LT, GT, EQUALS, LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%nonassoc       IN, RANGE;
%left           U, APPEND;
%left           PLUSMARK, MINUSMARK, PLUS, MINUS;
```

```
%left              STARMARK, SLASH, MUL, DIV, MOD;
%left              UMINUS;
%left              EXP;
%left              '$', LBRACK, '(', '{', DOTMARK, DOT, WHERE;
%left              STAR;

%%
!attribute declarations
!Terminals First

BIND, ARROW, IF, THEN, ELSE, ALL, SOME, NUMBER, SUM, PRODUCT, SET, MAXIMUM,
MINIMUM, UNION, INTERSECTION, SUCH, ELSE_IF, AS, CHOOSE, CONCEPT, DEFINITION,
DELAY, DO, END, EXCEPTION, EXPORT, FI, FOREACH, FROM, FUNCTION, GENERATE, HIDE,
IMPORT, INHERIT, INITIALLY, INSTANCE, INVARIANT, MACHINE, MESSAGE, MODEL, OD, OF,
OPERATOR, OTHERWISE, PERIOD, RENAME, REPLY, SEND, STATE, TEMPORAL, TIME, TO,
TRANSACTION, TRANSITION, TYPE, VALUE, VIRTUAL, WHEN, WHERE {
   %line : int;
}

QUESTION_MARK, LT, GT, EQUALS, PLUSMARK, MINUSMARK, STARMARK, SLASH,
DOTMARK, LBRACK {
   %line : int;
   %text : string;
}



NOT, AND, OR, IMPLIES, IFF, LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV,
MOD, EXP, U, APPEND, IN, RANGE {
    %text : string;
    %line : int;
}

INTEGER_LITERAL, REAL_LITERAL, CHAR_LITERAL, STRING_LITERAL {
    %text : string;
    %line : int;
}


NAME {
    %text : string;
    %line : int;
}

!Now Nonterminals.

spec {
    mod_types_s : string->string->string;
    global_type_s: string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    IP_STBL_INFO;
    ip_mcmxref_i : string->string;
    ip_lclzd_mcmxref_s : string->string->string;
    STBL_INFO;

    error_msgs_s :string;
}

module {
```

```
    mod_types_s : string->string->string;
    global_type_s: string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    ip_lclzd_mcmxref_s : string->string->string;
    ip_mcmxref_i : string->string;

    error_msgs_s : string;
}

function {
    module_name_s  : string;
    mxref_value_s : int;
    mod_types_s : string->string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    IP_MCMXREF_TBLS;

    error_msgs_s : string;

}

machine {
    module_name_s  : string;
    mxref_value_s : int;
    mod_types_s : string->string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    IP_MCMXREF_TBLS;

    error_msgs_s : string;
}

type {
    module_name_s  : string;
    mxref_value_s : int;
    mod_types_s : string->string->string;
    global_type_s: string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    IP_MCMXREF_TBLS;

    error_msgs_s  : string;
}
```

```
definition {
    module_name_s  : string;
    mxref_value_s : int;
    mod_types_s : string->string->string;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    IP_MCMXREF_TBLS;

    error_msgs_s : string;
}

instance {
    module_name_s : string;
    mxref_value_s : int;
    type_table_i : string->string->string;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;
    IP_MCMXREF_TBLS;

    error_msgs_s : string;
    d_error_s : string;
}

interface {
    module_name_s  : string;
    mxref_value_s : int;
    type_table_i : string->string->string;
    env_i : int;

    VISIBILITY_TBLS;

    ip_mxref_s : string->string;
    ip_mxref_i : string->string;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
    d_error_s : string;
}

inherits {

    error_msgs_s : string;

}

hide {

     error_msgs_s : string;
}

renames {

      error_msgs_s : string;
```

```
}

imports {
    type_table_i : string->string->string;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}

export {

    error_msgs_s : string;
}



messages {
    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}

message {
    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}

response {
    xref_value_i : int;
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}

response_cases {
    xref_value_i : int;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
```

```
}

response_body {
    xref_value_i : int;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


choose {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


reply {
    xref_value_i : int;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


sends {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


send {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}


transition {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
```

```
    STBL_INFO;

    error_msgs_s : string;
}


formal_message {
    xref_value_s : int;
    message_name_s : string;
    message_fargs_s : string;

    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    d_error_s : string;
    error_msgs_s : string;
}

actual_message {
    actual_text_s : string;
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}


where {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}


optionally_virtual {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}


optional_exception {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;
```

```
    error_msgs_s : string;
}



operator {
    xref_value_i : int;
    message_fargs_i : string;
    line_s : int;

    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}



optional_foreach {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


foreach {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}



concepts {
    module_name_i  : string;
    local_types_s : string->string;
    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}

concept {
    local_types_s : string->string;
    module_name_i  : string;
    xref_value_s : int;
```

```
    VISIBILITY_TBLS;

    IP_MCMXREF_TBLS;
    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
    d_error_s : string;
}


model {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


state {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}


invariant {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


initially {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


transactions {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}
```

```
transaction {
    d_error_s : string;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


action_list {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;

}



action {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


alternatives {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


guard {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;


}


temporals {
    VISIBILITY_TBLS;

    IP_STBL_INFO;
```

```
    STBL_INFO;

    error_msgs_s : string;
}


temporal {
    xref_value_s : int;
    d_error_s : string;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


optional_formal_name {
    name_text_s : string;
    name_params_s : string;
    args_i : string;
    env_i : int;
    xref_value_s : int;
    line_s : int;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
}


formal_name {
    name_text_s: string;
    name_params_s : string;
    env_i : int;
    xref_value_s : int;
    args_i : string;
    line_s : int;
    signature_s : string;

    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s : string;
    d_error_s : string;

}

formal_parameters {
    name_params_s : string;
```

```
    VISIBILITY_TBLS;

    IP_STBL_INFO;
    STBL_INFO;

    error_msgs_s  : string;
}


formal_arguments {
    name_fargs_s : string;
    args_text_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


field_list {
    fieldpattern_s : string;
    text_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}

field {
    fieldpattern_s : string;
    text_s : string;
    xref_value_s : int;
    d_error_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;
    error_msgs_s  : string;
}

declname_list {
    name_type_text_i : string;
    name_type_value_i : string;
    fieldpattern_s : string;
    xref_value_s : int;
    text_s : string;
    d_error_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
```

```
}


type_spec {
    type_name_text_s : string;
    type_name_value_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
    tmp_msg : string;
}


name_list {
    error_msgs_s : string;
}

optional_actual_name {
    full_name_s : string;
    actual_params_s : string;
    actual_name_text_s : string;
    line_s : int;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


actual_name {
    actual_name_text_s : string;
    full_name_s : string;
    actual_params_s : string;
    line_s : int;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


actual_parameters {
    actual_params_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}
```

```
actual_arguments {
    full_args_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


value_arguments {
    xref_value_i : int;


    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


arg_list {
    arglist_text_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


arg {
    arg_text_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


expression_list {
    xten_type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}
```

```
expression {
    xten_type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


middle_cases {
    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


restriction {
    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


literal {
    xten_type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


expressions {
    xten_type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}
```

```
pair_list {
    xten_type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}


pair {
    xten_type_s : string;
    type_s : string;

    IP_STBL_INFO;
    STBL_INFO;

    VISIBILITY_TBLS;

    error_msgs_s : string;
}



operator_list {
     xref_value_i : int;
     message_fargs_i : string;
     line_s : int;

     STBL_INFO;

     IP_MCMXREF_TBLS;

     d_error_s : string;
     error_msgs_s : string;
}

operator_symbol {
    operator_text_s : string;
    line_s : int;
}



%%
! productions of the grammar

start
        : spec
          {
            $1.type_table_i = $1.mod_types_s +|
                              { (GLOBAL_TYPE_NAMES : $1.global_type_s) };

            $1.ip_stbl_class_i = {(? : int : FALSE) };
            $1.ip_stbl_names_i = {(? : int : NULL_STRING) };
            $1.ip_stbl_params_i = {(? : int : NULL_STRING) };
            $1.ip_stbl_result_i = {(? : int : NULL_STRING) };
```

```
        $1.stbl_names_i = $1.ip_stbl_names_s;
        $1.stbl_class_i = $1.ip_stbl_class_s;
        $1.stbl_params_i = $1.ip_stbl_params_s;
        $1.stbl_result_i = $1.ip_stbl_result_s;


        $1.ip_mcmxref_i = $1.ip_mxref_s;
        $1.stbl_i = $1.ip_lclzd_mcmxref_s;


        ! test values
        %output("TYPES\n1. Global\n");
        %smaps_dump($1.type_table_i (GLOBAL_TYPE_NAMES));
        %output("2. From module test1\n");
        %smaps_dump($1.type_table_i ("test1"));
        %output("3. From module test4\n");
        %smaps_dump($1.type_table_i ("test4"));
        %output("4. From module test7\n");
        %smaps_dump($1.type_table_i ("test7"));
        %output("\n\nMODULE XREF \nModule Test1\n");
        %smaps_dump($1.stbl_i("test1"));
        %output("Module test4\n");
        %smaps_dump($1.stbl_i("test4"));
        %output("Module test7\n");
        %smaps_dump($1.stbl_i("test7"));

        ! dump xref info & error messages.
        %output("\n\n");
        %output("-----------------------------------------------\n");
        %output("-   Error Messages & Cross Reference Info     -\n");
        %output("-----------------------------------------------\n");
        %output($1.error_msgs_s);
        %output("\n\nCross Reference to Names.\n");
        %xrefs_dump($1.stbl_names_i);
        %output("\n\nCross Reference to Name Classes.\n");
        %xrefi_dump($1.stbl_class_i);
        %output("\n\nCross Reference to Name Parameters.\n");
        %xrefs_dump($1.stbl_params_i);
        %output("\n\nCross Reference to Name Results. \n");
        %xrefs_dump($1.stbl_result_i);
      }
    ;

spec
    : spec module
      {
        catmapup2_1($1,$2,mod_types);
        catmapup2_1($1,$2,global_type);

        !module names.
        passovr_1($1, $2, ip_mxref);
        passup_1($2, ip_mxref);

        !symbol table
        stbl_build2($1,$2);
        passdn_stbl2($1, $2);

        ! pass down needed type translations.
         passdn2_1($1, $2, type_table);

         passdn2_1($1, $2, ip_mcmxref);
```

```
            catmapup2_1($1, $2, ip_lclzd_mcmxref);

            $$.error_msgs_s = [$1.error_msgs_s,  $2.error_msgs_s];

          }
        |
          {
            $$.mod_types_s = {(? : string : {(? : string : UNDEFINED_TYPE)})};
            $$.global_type_s = {(? : string : UNDEFINED_TYPE) };

            ! module names & features.
            $$.ip_mxref_s = {(? : string : NULL_STRING)};
            $$.ip_lclzd_mcmxref_s = {(? : string : {(? : string : NULL_STRING)})};

            !symbol table
            stbl_build0();

            $$.error_msgs_s = "";

          }
        ;
          ! A production with nothing after the "|" means the empty string
          ! is a legal replacement for the left hand side.

module
        : function
          {
            passup_1($1, mod_types);
            $$.global_type_s = {(? : string : UNDEFINED_TYPE) };

            passdn_1($1, type_table);

            !symbol table
            passio1_1($1, ip_mxref);
            stbl_build1($1);
            passdn_stbl1($1);

            passdn_1($1, ip_mcmxref);
            $$.ip_lclzd_mcmxref_s = {($1.module_name_s : $1.ip_mcmxref_s)};


            passup_1($1, error_msgs);
          }
        | machine
          {
            passup_1($1, mod_types);
            $$.global_type_s = {(? : string : UNDEFINED_TYPE) };
            passdn_1($1, type_table);


            !symbol table
            passio1_1($1, ip_mxref);
            stbl_build1($1);
            passdn_stbl1($1);

            passdn_1($1, ip_mcmxref);
            $$.ip_lclzd_mcmxref_s = {($1.module_name_s : $1.ip_mcmxref_s)};

            passup_1($1, error_msgs);
          }
```

```
        | type
          {
            passup_2($1, mod_types, global_type);
            passdn_1($1, type_table);


            !symbol table
            passio1_1($1, ip_mxref);
            stbl_build1($1);
            passdn_stbl1($1);


            passdn_1($1, ip_mcmxref);
            $$.ip_lclzd_mcmxref_s = {($1.module_name_s : $1.ip_mcmxref_s)};

            passup_1($1, error_msgs);
          }
        | definition
          {
            passup_1($1, mod_types);
            $$.global_type_s = {(? : string : UNDEFINED_TYPE) };
            passdn_1($1, type_table);


            !symbol table
            passio1_1($1, ip_mxref);
            stbl_build1($1);
            passdn_stbl1($1);

            passdn_1($1, ip_mcmxref);
            $$.ip_lclzd_mcmxref_s = {($1.module_name_s : $1.ip_mcmxref_s)};

            passup_1($1, error_msgs);
          }
        | instance        ! of a generic module
          {
            $$.mod_types_s = {(? : string : {(? : string : UNDEFINED_TYPE)})};
            $$.global_type_s = {(? : string : UNDEFINED_TYPE) };
            passdn_1($1, type_table);


            !symbol table
            passio1_1($1, ip_mxref);
            stbl_build1($1);
            passdn_stbl1($1);


            passdn_1($1, ip_mcmxref);
            $$.ip_lclzd_mcmxref_s = {($1.module_name_s : $1.ip_mcmxref_s)};

            passup_1($1, error_msgs);
          }

        ;

function
        : optionally_virtual FUNCTION interface messages concepts END
          {
            passup_2($3,module_name, mxref_value);
            passovr_1($3, $5, module_name);
            $3.env_i = FUNCTION_CLASS;
```

```
            $$.mod_types_s = {($3.module_name_s : $5.local_types_s)};

            passdn_1($3, type_table);
            passio2_1($4,$5,ip_mcmxref);

            !symbol table
            passio1_1($3, ip_mxref);
            stbl_build3($3, $4, $5);
            passdn_stbl3($3, $4, $5);


            !visibility information
            passovr2x_2($3, $4, $5, visible_types, visible_names);

            $$.error_msgs_s = [$3.error_msgs_s, $4.error_msgs_s, $5.error_msgs_s];
          }
        ;
          ! Virtual modules are for inheritance only, never used directly.

machine
        : optionally_virtual MACHINE interface state messages transactions temporals
concepts END
          {
            passup_2($3,module_name, mxref_value);
            passovr_1($3, $8, module_name);
            $3.env_i = MACHINE_CLASS;
            $$.mod_types_s = {($3.module_name_s : $8.local_types_s)};

            passdn_1($3, type_table);
            passio2_1($5, $8,ip_mcmxref);

            !symbol table
            passio1_1($3, ip_mxref);
            stbl_build6($3, $4, $5, $6, $7, $8);
            passdn_stbl6($3, $4, $5, $6, $7, $8);

            !visibility information.
            passovr_2($3,$4,visible_types, visible_names);
            passovr2x_2($4,$5,$6,visible_types, visible_names);
            passovr2x_2($4,$7,$8, visible_types, visible_names);

            $$.error_msgs_s = [$3.error_msgs_s, $5.error_msgs_s, $8.error_msgs_s];
          }
        ;

type
        : optionally_virtual TYPE interface model messages transactions temporals concepts
END
      {
            passup_2($3,module_name, mxref_value);
            passovr_1($3, $8, module_name);
            $3.env_i = TYPE_CLASS;
            $$.mod_types_s = {($3.module_name_s : $8.local_types_s )} ;
            $$.global_type_s = {($3.module_name_s : $3.module_name_s),
                                (? : string : UNDEFINED_TYPE) };

            passdn_1($3, type_table);
            passio2_1($5, $8, ip_mcmxref);

            !symbol table
            passio1_1($3, ip_mxref);
```

```
            stbl_build6($3, $4, $5, $6, $7, $8);
            passdn_stbl6($3, $4, $5, $6, $7, $8);


            !visibility information.
            passovr_2($3,$4,visible_types, visible_names);
            passovr2x_2($4,$5,$6,visible_types, visible_names);
            passovr2x_2($4,$7,$8, visible_types, visible_names);

            $$.error_msgs_s = [$3.error_msgs_s, $5.error_msgs_s, $8.error_msgs_s];
          }
        ;

definition
        : DEFINITION interface concepts END
          {
            passup_2($2,module_name, mxref_value);
            passovr_1($2, $3, module_name);
            $2.env_i = DEFINITION_CLASS;
            $$.mod_types_s = {($2.module_name_s : $3.local_types_s)};

            passdn_1($2, type_table);
            passiol_1($3, ip_mcmxref);

            !symbol table
            passiol_1($2, ip_mxref);
            stbl_build2($2, $3);
            passdn_stbl2($2, $3);

            !visibility information.
            passovr_2($2, $3, visible_types, visible_names);

            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];
          }
        ;

instance
        : INSTANCE formal_name EQUALS actual_name END
          {
            $$.module_name_s = $2.name_text_s;
            $2.env_i = INSTANCE_CLASS;
            $$.mxref_value_s = $2.xref_value_s;


            passio0_1(ip_mcmxref);

            !symbol table
            $$.d_error_s = check_simple_decl($$.ip_mxref_i, $$.module_name_s, $2.line_s,
                                             $$.stbl_names_i);
            mk_simple_decl_io($$.ip_mxref, $$.d_error_s, $$.module_name_s,
                              $$.mxref_value_s);
            stbl_build2($2, $4);
            passdn_stbl2($2, $4);

            !visibility information.
            $2.visible_types_i = $$.type_table_i (GLOBAL_TYPE_NAMES);
            $2.visible_names_i = $$.stbl_i($$.module_name_s);
            passovr_2($2, $4, visible_types, visible_names);

            $$.error_msgs_s = $$.d_error_s;
          }
```

```
        | INSTANCE foreach actual_name END
          {
            ! Check this entire section of code for interfaces...
            $$.module_name_s = $3.actual_name_text_s;   !check this w/ Prof. B.
            $$.mxref_value_s = get_new_xref($3.actual_name_text_s);
            $$.d_error_s = check_simple_decl($$.ip_mxref_i, $$.module_name_s, $3.line_s,
                                             $$.stbl_names_i);
            mk_simple_decl_io($$.ip_mxref, $$.d_error_s, $$.module_name_s,
                              $$.mxref_value_s);

            passio0_1(ip_mcmxref);

            !symbol table
            stbl_build2($2,$3);
            passdn_stbl2($2,$3);

            !visibility information
            $2.visible_types_i = $$.type_table_i (GLOBAL_TYPE_NAMES);
            $2.visible_names_i = $$.stbl_i ($$.module_name_s);
            passovr_2($2, $3, visible_names, visible_types);

            $$.error_msgs_s = $$.d_error_s;
          }
        ;
          ! For making instances or partial instantiations of generic modules.
          ! The foreach clause allows defining sets of instances.

interface
        : formal_name inherits imports export
          {
            $$.module_name_s = $1.name_text_s;
            $$.mxref_value_s = $1.xref_value_s;
            $1.args_i = "";
            passdn_1($1, env);
            $$.d_error_s = check_simple_decl($$.ip_mxref_i, $$.module_name_s, $1.line_s,
                                             $$.stbl_names_i);
            mk_simple_decl_io($$.ip_mxref, $$.d_error_s, $$.module_name_s,
                              $$.mxref_value_s);


            !symbol table
            stbl_build2($1, $3);
            passdn_stbl2($1, $3);


            !visibility information
            $1.visible_types_i = $$.type_table_i (GLOBAL_TYPE_NAMES);
            $1.visible_names_i = $$.stbl_i ($$.module_name_s) +|
                                 {(CURRENT_MODULE_TAG : $$.module_name_s)};
            $3.visible_types_i = $1.visible_types_s +|
                                 $$.type_table_i ($$.module_name_s);
            passovr_1($1, $3, visible_names);
            passup_2($3, visible_types, visible_names);


            $$.error_msgs_s = [$$.d_error_s, $1.error_msgs_s];
          }
        ;
          ! This part describes the static aspects of a module's interface.
          ! The dynamic aspects of the interface are described in the messages.
          ! A module is generic iff it has parameters.
```

```
          ! The parameters can be constrained by a WHERE clause.
          ! A module can inherit the behavior of other modules.
          ! A module can import concepts from other modules.
          ! A module can export concepts for use by other modules.

inherits
        : inherits INHERIT actual_name hide renames
          { }
        |
          { }
        ;
          ! Ancestors are generalizations or simplified views of a module.
          ! A module inherits all of the behavior of its ancestors.
          ! Hiding a message or concept means it will not be inherited.
          ! Inherited components can be renamed to avoid naming conflicts.

hide
        : HIDE name_list
          { }
        |
          { }
        ;
          ! Useful for providing limited views of an actor.
          ! Different user classes may see different views of a system.
          ! Messages and concepts can be hidden.

renames
        : renames RENAME NAME AS NAME
          { }
        |
          { }
        ;
          ! Renaming is useful for preventing name conflicts when inheriting
          ! from multiple sources, and for adapting modules for new uses.
          ! The parameters, model and state components, messages, exceptions,
          ! and concepts of an actor can be renamed.

imports
        : imports IMPORT name_list FROM actual_name
          {

             !visibility information.
             passio0_2(visible_types, visible_names);

             !for now -- until importation implemented.
             stbl_build1($1);
             passdn_stbl1($1);
          }
        |
          {

             !visibility information.
             passio0_2(visible_types, visible_names);

             !symbol table
             stbl_build0();
          }
        ;

export
        : EXPORT name_list
```

```
        { }
      |
        { }
      ;

messages
      : messages message
        {
           passio2_1($1, $2, ip_mcmxref);

           !symbol table
           stbl_build2($1, $2);
           passdn_stbl2($1, $2);

           !visibility information
           passdn2_2($1,$2, visible_types, visible_names);

           !error messages
           $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
      |
        {
           passio0_1(ip_mcmxref);

           !symbol table
           stbl_build0();


           !error messages
           $$.error_msgs_s = "";
        }
      ;

message
      : MESSAGE formal_message operator response
        {
           passio2_1($2, $3, ip_mcmxref);

           passovr_2($2, $3, xref_value, message_fargs);
           passovr_1($2, $4, xref_value);

           !symbol table
           stbl_build3($2, $3, $4);
           passdn_stbl3($2, $3, $4);

           !visibility information
           passdn_2($2, visible_types, visible_names);
           passovr_2($2, $4, visible_types, visible_names);

           !error messages
           $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s, $4.error_msgs_s];
        }
      ;

response
      : response_body
        {
            passdn_1($1, xref_value);

            !symbol table
            stbl_build1($1);
```

```
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        | response_cases
          {
            passdn_1($1, xref_value);

            !symbol table
            stbl_buildl($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        ;

response_cases
        : WHEN expression_list response_body response_cases
          {
            passdn2_1($3,$4, xref_value);

            !symbol table
            stbl_build3($2, $3, $4);
            passdn_stbl3($2, $3, $4);

            !visibility information
            passdn3_2($2, $3, $4, visible_types, visible_names);

            !error_messages
            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s, $4.error_msgs_s];
          }
        | OTHERWISE response_body
          {
            passdn_1($2, xref_value);

            !symbol table
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);

          }
        ;

response_body
        : choose reply sends transition
          {
             passdn_1($2, xref_value);

             ! symbol table
```

```
            stbl_build4($1, $2, $3, $4);
            passdn_stbl4($1, $2, $3, $4);

            !visibility information
            passdn_2($1, visible_types, visible_names);
            passovr3x_2($1, $2, $3, $4, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s, $3.error_msgs_s,
                               $4.error_msgs_s];
        }

    ;

choose
    : CHOOSE '(' field_list restriction ')'
        {
            !symbol table
            stbl_build2($3, $4);
            passdn_stbl2($3, $4);

            !visibility information
            passiol_2($3, visible_types, visible_names);
            passovr_2($3, $4, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$3.error_msgs_s, $4.error_msgs_s];
        }
    |
        {
            !symbol table
            stbl_build0();

            !visibility information
            passio0_2(visible_types, visible_names);

            !error messages
            $$.error_msgs_s = "";
        }
    ;
reply
    : REPLY actual_message where
        {
            !symbol table
            passio2_3($2, $3, ip_stbl_class, ip_stbl_names, ip_stbl_params);
            $2.ip_stbl_result_i = $$.ip_stbl_result_i +|
                                  {($$.xref_value_i :
                                    $2.actual_text_s)};
            passovr_1($2, $3, ip_stbl_result);
            passup_1($3, ip_stbl_result);
            passdn_stbl2($2, $3);

            !visibility information
            passdn2_2($2, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];
        }
    | GENERATE actual_message where       ! used in generators
        {
            !symbol table
```

```
            passio2_3($2, $3, ip_stbl_class, ip_stbl_names, ip_stbl_params);
            $2.ip_stbl_result_i = $$.ip_stbl_result_i +|
                                  {($$.xref_value_i : $2.actual_text_s)};
            passovr_1($2, $3, ip_stbl_result);
            passup_1($3, ip_stbl_result);
            passdn_stbl2($2, $3);

            !visibility information
            passdn2_2($2, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];

        }
    |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
    ;

sends
    : sends send
        {
            stbl_build2($1, $2);
            passdn_stbl2($1, $2);

            !visibility information
            passdn2_2($1, $2, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
    |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
    ;

send
    : optional_foreach SEND actual_message TO actual_name where
        {
            stbl_build4($1, $3, $5, $6);
            passdn_stbl4($1, $3, $5, $6);

            !visibility information
            passdn4_2($1, $3, $5, $6, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$3.error_msgs_s, $5.error_msgs_s, $6.error_msgs_s];
        }
    ;

transition
    : TRANSITION expression_list        ! for describing state changes
        {
```

```
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
      |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
      ;

formal_message
      : optional_exception optional_formal_name formal_arguments
        {
           $$.message_name_s = $2.name_text_s;
           $$.message_fargs_s = $3.name_fargs_s;
           $2.args_i = $3.args_text_s;
           $2.env_i = MESSAGE_CLASS;
           passup_1($2, xref_value);

           mk_complex_decl($$.ip_mcmxref, $$.d_error_s, $2.name_text_s,
                           $2.xref_value_s, $3.name_fargs_s);


           !symbol table
           stbl_build2($2, $3);
           passdn_stbl2($2, $3);


           !visibility information
           passio2_2($2, $3, visible_types, visible_names);

           !error messages
           $$.d_error_s = check_complex_decl($$.ip_mcmxref_i, $2.name_text_s,
                          $3.name_fargs_s, $2.line_s, $$.stbl_names_i);
           $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];
        }
      ;

actual_message
      : optional_exception optional_actual_name formal_arguments
        {
            $$.actual_text_s = ($2.full_name_s == "")
                             -> $3.name_fargs_s
                            #  ($3.name_fargs_s == "")
                               -> $2.full_name_s
                               #  [$2.full_name_s, "(",
                                  $3.name_fargs_s,")"];


            stbl_build2($2, $3);
            passdn_stbl2($2, $3);

            !visibility information
```

```
            passdn2_2($2, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];
        }
    ;

where
    : WHERE expression_list
        {
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
    |       %prec SEMI        ! must have a lower precedence than WHERE
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
    ;

optionally_virtual
    : VIRTUAL
        { }
    |
        { }
    ;

optional_exception
    : EXCEPTION
        { }
    |       %prec SEMI
        { }
    ;

operator
    : OPERATOR operator_list
        {
            passdn_2($2, xref_value, message_fargs);
            passiol_1($2, ip_mcmxref);

            !symbol table
            stbl_build0();
            passdn_stbl1($2);

            !error messages
            passup_1($2, error_msgs);
        }
    |
        {
            passio0_1(ip_mcmxref);

            !symbol table
            stbl_build0();
```

```
            $$.error_msgs_s = "";
        }
    ;

optional_foreach
    : foreach
        {
            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
        }
    |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";

        }
    ;

foreach
    : FOREACH '(' field_list restriction ')'
        {
            stbl_build2($3, $4);
            passdn_stbl2($3, $4);

            !visibility information
            passio2_2($3, $4, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$3.error_msgs_s, $4.error_msgs_s];
        }
    ;
        ! foreach is used to describe a set of messages or instances

concepts
    : concepts concept
        {
          passdn2_1($1, $2, module_name);
          catmapup2_1($1,$2,local_types);
          passio2_1($1,$2, ip_mcmxref);

          !symbol table
          stbl_build2($1, $2);
          passdn_stbl2($1, $2);


          !visibility information
          passdn2_2($1, $2, visible_types, visible_names);

          !error messages
          $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
    |
```

```
        {
          $$.local_types_s = {(? : string : UNDEFINED_TYPE)};
          passio0_1(ip_mcmxref);

          !symbol table
          stbl_build0();


          !errror messages
          $$.error_msgs_s = "";
        }
      ;

concept
      : CONCEPT formal_name ':' type_spec where
        ! constants
        {

          $$.local_types_s = ($4.type_name_text_s == SPEC_LIBRARY_MODULE_type)
              -> { ($2.name_text_s : [$2.name_text_s, "@", $$.module_name_i]) }
              #
                 { (? : string : UNDEFINED_TYPE ) };

          passup_1($2, xref_value);
          $$.d_error_s = check_simple_decl($$.ip_mcmxref_i, $2.name_text_s, $2.line_s,
                         $$.stbl_names_i);
          mk_simple_decl_io($$.ip_mcmxref, $$.d_error_s, $2.name_text_s,
                           $2.xref_value_s);

          !symbol table
          $2.args_i = "";
          $2.env_i = CONCEPT_CLASS1;
          stbl_build3($2, $4, $5);
          passdn_stbl3($2, $4, $5);


          !visibility information
          passdn_2($2, visible_types, visible_names);
          passovr2x_2($2, $4, $5, visible_types, visible_names);

          !error messages  -- incomplete for now...
          $$.error_msgs_s = [$$.d_error_s, $2.error_msgs_s, $4.error_msgs_s,
                             $5.error_msgs_s];



        }
      | CONCEPT formal_name formal_arguments where VALUE value_arguments where
        ! functions, defined with preconditions and postconditions
        {
          $$.local_types_s = { (? : string : UNDEFINED_TYPE) };

          passup_1($2, xref_value);
          $$.d_error_s = check_complex_decl($$.ip_mcmxref_i, $2.name_text_s,
                         $3.name_fargs_s, $2.line_s, $$.stbl_names_i);
          mk_complex_decl($$.ip_mcmxref, $$.d_error_s, $2.name_text_s,
                          $2.xref_value_s, $3.name_fargs_s);

          !symbol table
          $2.args_i = $3.args_text_s;
          $2.env_i = CONCEPT_CLASS2;
```

```
            passovr_1($2, $6, xref_value);
            stbl_build5($2, $3, $4, $6, $7);
            passdn_stbl5($2, $3, $4, $6, $7);


            !visibility information.
            passdn_2($2, visible_types, visible_names);
            passovr_2($2, $3, visible_types, visible_names);
            passovr2x_2($3,$4,$6, visible_types, visible_names);
            passovr_2($6, $7, visible_types, visible_names);

            !error messages -- incomplete for now...
            $$.error_msgs_s = [$$.d_error_s, $2.error_msgs_s, $3.error_msgs_s,
                               $4.error_msgs_s, $6.error_msgs_s, $7.error_msgs_s];
          }
        ;


value_arguments     ! a new nonterminal to simplify equations.
         : formal_arguments
           {
              !symbol table
              passiol_3($1, ip_stbl_class, ip_stbl_names, ip_stbl_params);
              passdn_1($1, ip_stbl_result);
              $$.ip_stbl_result_s = ($1.args_text_s == "")
                                  ->  $1.ip_stbl_result_s
                                  #  $1.ip_stbl_result_s +|
                                      {($$.xref_value_i :
                                        [ "(", $1.args_text_s, ")" ] )};


              passdn_stbl1($1);

              !visibility information
              passiol_2($1, visible_types, visible_names);

              !error_messages
              passup_1($1, error_msgs);
           }

model                ! data types have conceptual models for values
        : MODEL formal_arguments invariant
          {
              stbl_build2($2, $3);
              passdn_stbl2($2, $3);


              !visibility information
              passiol_2($2, visible_types, visible_names);
              passovr_2($2,$3, visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s];
          }
        ;

state                ! machines have conceptual models for states
        : STATE formal_arguments invariant initially
          {
              stbl_build3($2, $3, $4);
              passdn_stbl3($2, $3, $4);
```

```
            !visibility information
            passiol_2($2, visible_types, visible_names);
            passovr2x_2($2,$3,$4, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$2.error_msgs_s, $3.error_msgs_s, $4.error_msgs_s];
        }
    ;

invariant       ! invariants are true for all states or instances
    : INVARIANT expression_list
        {
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
    ;

initially       ! initial conditions are true only at the beginning
    : INITIALLY expression_list
        {
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2,visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
    ;

transactions
    : transactions transaction
        {
            stbl_build2($1, $2);
            passdn_stbl2($1, $2);

            !visibility information
            passio2_2($1,$2,visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
    |
        {
            stbl_build0();
            passio0_2(visible_types, visible_names);

            !error messages
            $$.error_msgs_s = "";
        }
    ;

transaction
```

```
        : TRANSACTION formal_name EQUALS action_list where
          {
              $2.args_i = "";
              $2.env_i = TRANSACTION_CLASS;

              !symbol table
              stbl_build3($2, $4, $5);
              passdn_stbl3($2, $4, $5);

              !visibility information
              passdn_1($2,visible_types);
              $$.d_error_s = check_simple_decl($$.visible_names_i, $2.name_text_s,
                             $2.line_s, $$.stbl_names_i);
              mk_simple_decl($$.visible_names_i, $$.d_error_s, $2.name_text_s,
                             $2.xref_value_s, $$.visible_names_s);
              passovr2x_1($2, $4, $5, visible_types);
              passovr2x_1($$, $4, $5, visible_names);

              !error messages
              $$.error_msgs_s = [$2.error_msgs_s, $4.error_msgs_s, $5.error_msgs_s];

          }
        ;
          ! Transactions are atomic.
          ! The where clause can specify timing constraints.

action_list
        : action_list ';' action    %prec SEMI    ! sequence
          {
              stbl_build2($1, $3);
              passdn_stbl2($1, $3);

              !visibility information
              passdn2_2($1, $3,visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
          }
        | action
          {
              stbl_build1($1);
              passdn_stbl1($1);

              !visibility information
              passdn_2($1, visible_types, visible_names);

              !error_messages
              passup_1($1, error_msgs);
          }
        ;

action
        : action action    %prec STAR    ! unordered set of actions
          {
              stbl_build2($1, $2);
              passdn_stbl2($1, $2);

              !visibility information
              passdn2_2($1, $2, visible_types, visible_names);

              !error messages
```

```
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
          }
        | IF alternatives FI    ! choice
          {
              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }

        | DO alternatives OD    ! repeated choice
          {
              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }
        | actual_name    ! a normal message or subtransaction
          {
              stbl_build1($1);
              passdn_stbl1($1);

              !visibility information
              passdn_2($1, visible_types, visible_names);

              !error_messages
              passup_1($1, error_msgs);
          }
        | EXCEPTION actual_name    ! an exception message
          {

              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }
        ;

alternatives
        : alternatives OR guard action_list
          {
              stbl_build3($1, $3, $4);
              passdn_stbl3($1, $3, $4);

              !visibility information
              passdn3_2($1, $3, $4, visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s, $4.error_msgs_s];
```

```
        }
      | guard action_list
        {
            stbl_build2($1, $2);
            passdn_stbl2($1, $2);

            !visibility information
            passdn2_2($1, $2, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
      ;

guard
      : WHEN expression ARROW
        {
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
      |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
      ;

temporals
      : temporals temporal
        {
            stbl_build2($1, $2);
            passdn_stbl2($1, $2);

            !visibility information
            passio2_2($1, $2, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
        }
      |
        {
            stbl_build0();
            passio0_2(visible_types, visible_names);

            !error messages
            $$.error_msgs_s = "";
        }
      ;

temporal
      : TEMPORAL NAME where response
        {
            $$.xref_value_s = get_new_xref($2.%text);
```

```
            $4.xref_value_i = $$.xref_value_s;

            !symbol table
            $3.ip_stbl_class_i = $$.ip_stbl_class_i +|
                                 {($$.xref_value_s : TEMPORAL_CLASS)};
            $3.ip_stbl_names_i = $$.ip_stbl_names_i +|
                                 {($$.xref_value_s : $2.%text)};
            passovr_2($3, $4, ip_stbl_class, ip_stbl_names);
            passup_2($4, ip_stbl_class, ip_stbl_names);
            passio2_2($3, $4, ip_stbl_params, ip_stbl_result);
            passdn_stbl2($3, $4);

            !visibility information
            passdn2_1($3, $4, visible_types);
            $$.d_error_s = check_simple_decl($$.visible_names_i, $2.%text,
                           $2.%line, $$.stbl_names_i);
            mk_simple_decl($$.visible_names_i, $$.d_error_s, $2.%text,
                           $$.xref_value_s, $$.visible_names_s);
            passovr2x_1($$, $3, $4, visible_names);

            !error messages
            $$.error_msgs_s = [$$.d_error_s, $3.error_msgs_s, $4.error_msgs_s];
          }
        ;
          ! Temporal events are trigged at absolute times,
          ! in terms of the local clock of the actor.
          ! The "where" describes the triggering conditions
          ! in terms of TIME, PERIOD, and DELAY.

optional_formal_name
        : formal_name
          {
              passup_3($1, name_params, name_text, line);
              passdn_2($1, args, env);
              passup_1($1, xref_value);

              !symbol table
              stbl_build1($1);
              passdn_stbl1($1);


              !visibility information
              passio1_2($1, visible_types, visible_names);


              !error messages
              passup_1($1, error_msgs);
          }
        |
          {
              $$.line_s = -1;
              $$.name_params_s = "";
              $$.name_text_s = "";
              $$.xref_value_s = get_new_xref($$.args_i);

              !symbol table
              add_elem($$.ip_stbl_class_i, $$.xref_value_s, $$.env_i, $$.ip_stbl_class_s);
              add_elem($$.ip_stbl_names_i, $$.xref_value_s,$$.args_i, $$.ip_stbl_names_s);
              passio0_2(ip_stbl_params, ip_stbl_result);
```

```
            !visibility information
            passio0_2(visible_types, visible_names);

            !error messages
            $$.error_msgs_s = "";

        }
      ;

formal_name
      : NAME formal_parameters
        {
          $$.line_s = $1.%line;
          $$.name_text_s = $1.%text;
          passup_1($2, name_params);
          $$.xref_value_s = get_new_xref($1.%text);

          !symbol table
          $$.signature_s = ( $$.args_i == "")
                         -> ""
                         #  ["(", $$.args_i, ")" ];
          add_elem($$.ip_stbl_class_i, $$.xref_value_s, $$.env_i, $2.ip_stbl_class_i);
          add_elem($$.ip_stbl_names_i, $$.xref_value_s,
                   $1.%text ^ $$.signature_s, $2.ip_stbl_names_i);
          add_elem($$.ip_stbl_params_i, $$.xref_value_s, $2.name_params_s,
                   $2.ip_stbl_params_i);
          passdn_1($2, ip_stbl_result);
          passup_4($2, ip_stbl_class, ip_stbl_names, ip_stbl_params, ip_stbl_result);
          passdn_stbl1($2);

          ! visibility information
          passio1_2($2, visible_types, visible_names);

          !error messages.
          passup_1($2, error_msgs);
        }
      ;

formal_parameters        ! parameter values are determined at specification time
      : '{' field_list '}' where
        {
           $$.name_params_s = $2.fieldpattern_s;

           !symbol table
           stbl_build2($2, $4);
           passdn_stbl2($2, $4);


           ! visibility information
           passio1_2($2, visible_types, visible_names);
           passovr_2($2, $4, visible_types, visible_names);

           !error messages.
           $$.error_msgs_s = [$2.error_msgs_s, $4.error_msgs_s];
        }
      |
        {
           $$.name_params_s = NULL_STRING;

           !symbol table
           stbl_build0();
```

```
            !visibility information
            passio0_2(visible_types, visible_names);

            ! error messages
            $$.error_msgs_s = "";
        }
    ;

formal_arguments        ! arguments are evaluated at run-time
    : '(' field_list ')'
        {
            $$.name_fargs_s = $2.fieldpattern_s;
            $$.args_text_s = $2.text_s;

            !symbol table
            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passio1_2($2, visible_types, visible_names);

            !error messages
            passup_1($2, error_msgs);
        }
    |
        {
            $$.name_fargs_s = "";
            $$.args_text_s = "";

            !symbol table
            stbl_build0();


            !visibility information
            passio0_2(visible_types, visible_names);

            !error messages
            $$.error_msgs_s = "";
        }
    ;

field_list
    : field_list ',' field
        {
            $$.fieldpattern_s = [$1.fieldpattern_s, ELEM_DELIMITER,
                        $3.fieldpattern_s];
            $$.text_s = [$1.text_s, ", ", $3.text_s];

            !symbol table
            stbl_build2($1, $3);
            passdn_stbl2($1, $3);

            !visibility information.
            passio2_2($1, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
        }
    | field
```

```
        {
           passup_2($1, fieldpattern, text);

           !symbol table
           stbl_build1($1);
           passdn_stbl1($1);

           !visibility information
           passiol_2($1, visible_types, visible_names);

           !error messages
           passup_1($1, error_msgs);
        }
      ;

field
      : declname_list ':' type_spec
        {
           $1.name_type_text_i = $3.type_name_text_s;
           $$.text_s = [$1.text_s, " : ", $3.type_name_text_s];

            $1.name_type_value_i = $3.type_name_value_s;
            passup_1($1, fieldpattern);

            !symbol table
            stbl_build2($1, $3);
            passdn_stbl2($1, $3);

            !visibility information.
            passiol_2($1, visible_types, visible_names);
            passdn_2($3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
        }
      | '$' NAME ':' type_spec
        {
           $$.fieldpattern_s = ["$", $2.%text, ":",$4.type_name_value_s ];
           $$.text_s = ["$", $2.%text, " : ", $4.type_name_text_s];

           !symbol table
           $$.xref_value_s = get_new_xref($2.%text);
           add_elem($$.ip_stbl_class_i, $$.xref_value_s, VARIABLE_CLASS,
                    $4.ip_stbl_class_i);
           add_elem($$.ip_stbl_names_i, $$.xref_value_s, "$"^$2.%text,
                    $4.ip_stbl_names_i);
           add_elem($$.ip_stbl_result_i, $$.xref_value_s, $4.type_name_value_s,
                    $4.ip_stbl_result_i);
           passdn_1($4, ip_stbl_params);
           passup_4($4, ip_stbl_class, ip_stbl_names, ip_stbl_result, ip_stbl_params);
           passdn_stbl1($4);


           !visibility information
           $$.visible_types_s = ($4.type_name_value_s == SPEC_LIBRARY_MODULE_type)
                        -> $$.visible_types_i +|
                           {($2.%text : [$2.%text, "@",
                            "__local" ])}
                        #  $$.visible_types_i;
           $$.d_error_s = check_simple_decl($$.visible_names_i, $2.%text,
                          $2.%line, $$.stbl_names_i);
```

```
            mk_simple_decl_io($$.visible_names, $$.d_error_s, $2.%text, $$.xref_value_s);
            passdn_1($4, visible_types);
            passovr_1($$, $4, visible_names);

            !error messages
            $$.error_msgs_s = [$$.d_error_s, $4.error_msgs_s];
          }
        | QUESTION_MARK
          {
            $$.fieldpattern_s = "?";
            $$.text_s = "?";

             !symbol table
             stbl_build0();


            !visibility information
            passio0_1(visible_types);


            !error messages
            $$.error_msgs_s = error_message(UNRESOLVED_TYPE, $1.%line);
          }
        ;

type_spec
        : actual_name        ! name of a data type
            {
              $$.type_name_text_s = $1.actual_name_text_s;
              $$.type_name_value_s = $$.visible_types_i ($1.actual_name_text_s);

              !symbol table
              stbl_build1($1);
              passdn_stbl1($1);




              !visibility information
              passdn_2($1, visible_types, visible_names);


              !error messages
              $$.tmp_msg = error_message(UNDECLARED_TYPE, $1.line_s,
                           $1.actual_name_text_s);
              $$.error_msgs_s = ($$.visible_types_i ($1.actual_name_text_s) ==
                                 UNDEFINED_TYPE)
                              -> $$.tmp_msg
                              # "";
            }
        | QUESTION_MARK
            {
              $$.type_name_text_s = "?";
              $$.type_name_value_s = "?";

              !symbol table
              stbl_build0();

              !error messages
              $$.error_msgs_s = error_message(UNRESOLVED_TYPE, $1.%line);
            }
```

```
        ;

! This structure was added so that a name list used for declarative
! purposes (e.g. in a field) could be easily distinguished from a name list
! used in an applicative structure (e.g. imports, export, hide).
! This lessened the actual attribute load of the name_list structure.
declname_list
        : declname_list NAME
          {
             $$.xref_value_s = get_new_xref($2. %text);
             $$.text_s = [$1.text_s, " ", $2.%text];
             passdn_2($1, name_type_value, name_type_text);
             $$.fieldpattern_s = [ $1.fieldpattern_s, ELEM_DELIMITER,
                                   $2.%text, ":", $$.name_type_value_i ];

             !symbol table
             passdn_4($1, ip_stbl_class, ip_stbl_names, ip_stbl_result, ip_stbl_params);
             add_elem($1.ip_stbl_class_s, $$.xref_value_s, VARIABLE_CLASS,
                      $$.ip_stbl_class_s);
             add_elem($1.ip_stbl_names_s, $$.xref_value_s, $2.%text, $$.ip_stbl_names_s);
             add_elem($1.ip_stbl_result_s, $$.xref_value_s, $$.name_type_value_i,
                      $$.ip_stbl_result_s);
             passup_1($1, ip_stbl_params);
             passdn_stbl1($1);

              !visibility information
              passdn_2($1, visible_types, visible_names);
              $$.d_error_s = check_simple_decl($1.visible_names_s, $2.%text,
                             $2.%line, $$.stbl_names_i);
              mk_simple_decl($1.visible_names_s, $$.d_error_s, $2.%text,
                             $$.xref_value_s, $$.visible_names_s);
              $$.visible_types_s = ($$.name_type_text_i == SPEC_LIBRARY_MODULE_type)
                                   -> $1. visible_types_s +|
                                        {($2.%text : [$2.%text, "@",
                                   "__local" ])}
                                     # $1.visible_types_s;

             $$.error_msgs_s = [$1.error_msgs_s, $$.d_error_s];
          }
        | NAME
          {
             $$.xref_value_s = get_new_xref($1. %text);
             $$.fieldpattern_s = [$1.%text, ":", $$.name_type_value_i];
             $$.text_s = $1.%text;

             !symbol table
             add_elem($$.ip_stbl_class_i, $$.xref_value_s, VARIABLE_CLASS,
                      $$.ip_stbl_class_s);
             add_elem($$.ip_stbl_names_i, $$.xref_value_s, $1.%text, $$.ip_stbl_names_s);
             add_elem($$.ip_stbl_result_i, $$.xref_value_s, $$.name_type_value_i,
                      $$.ip_stbl_result_s);
             passio0_1(ip_stbl_params);


              !visibility information
              ! -- eventually need to make local with module name...
              $$.visible_types_s = ($$.name_type_text_i == SPEC_LIBRARY_MODULE_type)
                                   -> $$. visible_types_i +|
                                   {($1.%text : [$1.%text, "@",
                                      "__local" ])}
                                    # $$.visible_types_i;
```

```
            $$.d_error_s = check_simple_decl($$.visible_names_i, $1.%text,
                           $1.%line, $$.stbl_names_i);
            mk_simple_decl_io($$.visible_names, $$.d_error_s, $1.%text,
                              $$.xref_value_s);

            $$.error_msgs_s = $$.d_error_s;
          }
        ;


name_list
        : name_list NAME
          { }
        | NAME
          { }
        ;


optional_actual_name
        : actual_name
          {
              passup_4($1, actual_name_text, actual_params, full_name, line);

              stbl_build1($1);
              passdn_stbl1($1);

              !visibility information
              passdn_2($1, visible_types, visible_names);

              !error_messages
              passup_1($1, error_msgs);
          }
         |
          {
              $$.line_s = -1;
              $$.actual_name_text_s = "";
              $$.actual_params_s = "";
              $$.full_name_s = "";

              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        ;

actual_name
        : NAME actual_parameters
          {
            $$.actual_name_text_s = $1.%text;
            $$.line_s = $1.%line;
            passup_1($2, actual_params);
            $$.full_name_s = ($2.actual_params_s == "")
                          -> $1.%text
                          # [$1.%text, "(", $2.actual_params_s, ")"];

            !symbol table
            stbl_build1($2);
            passdn_stbl1($2);
```

```
            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
          }
        ;

actual_parameters        ! parameter values are determined at specification time
        : '{' arg_list '}'
          {
             $$.actual_params_s = $2.arglist_text_s;

              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }
        |       %prec SEMI        ! must have a lower precedence than '{'
          {
              $$.actual_params_s = "";

              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        ;

actual_arguments        ! arguments are evaluated at run-time
        : '(' arg_list ')'
          {
             $$.full_args_s = ($2.arglist_text_s == "")
                              -> ""
                              # ["(", $2.arglist_text_s, ")"];

              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }
        |       %prec SEMI        ! must have a lower precedence than '('
          {
              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        ;

arg_list
        : arg_list ',' arg        %prec COMMA
          {
```

```
            $$.arglist_text_s = [$1.arglist_text_s, ACTUAL_DELIM, $3.arg_text_s];

            stbl_build2($1, $3);
            passdn_stbl2($1, $3);

            !visibility information
            passdn2_2($1, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
          }
        | arg
          {
            $$.arglist_text_s = $1.arg_text_s;

            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }

        ;

arg
        : expression
          {
            $$.arg_text_s = $1.xten_type_s;

            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        | pair
          {
            $$.arg_text_s = $1.xten_type_s;

            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        ;

expression_list
        : expression_list ',' expression        %prec COMMA
          {
            ! all types in an expression list must be the same
            $$.xten_type_s = $1.xten_type_s;
```

```
            stbl_build2($1, $3);
            passdn_stbl2($1, $3);

            !visibility information
            passdn2_2($1, $3, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
          }
        | expression        %prec COMMA
          {
            passup_1($1, xten_type);

            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        ;

expression
        : quantifier '(' field_list restriction BIND expression ')'
          {
            $$.xten_type_s = ":boolean";

            stbl_build3($3, $4, $6);
            passdn_stbl3($3, $4, $6);

            !visibility information
            passdn3_2($3, $4, $6, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$3.error_msgs_s, $4.error_msgs_s, $6.error_msgs_s];
          }
        | actual_name actual_arguments
          {
            $$.xten_type_s = [REF_SYMBOL, $1.full_name_s, $2.full_args_s];

            stbl_build2($1, $2);
            passdn_stbl2($1, $2);

            !visibility information
            passdn2_2($1, $2, visible_types, visible_names);

            !error messages
            $$.error_msgs_s = [$1.error_msgs_s, $2.error_msgs_s];
          }
        | actual_name '@' actual_name actual_arguments
          {
            $$.xten_type_s = [REF_SYMBOL, $1.full_name_s, "@", $3.full_name_s,
                              $4.full_args_s];


            stbl_build3($1, $3, $4);
            passdn_stbl3($1, $3, $4);
```

```
      !visibility information
      passdn3_2($1, $3, $4, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s, $4.error_msgs_s];
  }
| NOT expression          %prec NOT
  {
     $$.xten_type_s = [REF_SYMBOL, $1.%text, "(", $2.xten_type_s, ")"];

      stbl_build1($2);
      passdn_stbl1($2);

      !visibility information
      passdn_2($2, visible_types, visible_names);

      !error_messages
      passup_1($2, error_msgs);
  }
| expression AND expression          %prec AND
  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression OR expression          %prec OR
  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression IMPLIES expression          %prec IMPLIES
  {

      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
```

```
  }
| expression IFF expression        %prec IFF
  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression LT expression     %prec LE
  {
      $$.xten_type_s = [REF_SYMBOL, "<" , "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression GT expression     %prec LE
  {
      $$.xten_type_s = [REF_SYMBOL, ">", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression EQUALS expression     %prec LE
  {
      $$.xten_type_s = [REF_SYMBOL, "=", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression LE expression        %prec LE
  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];
```

```
        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression GE expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression NE expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression NLT expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression NGT expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
```

```
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    )
  | expression NLE expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression NGE expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression EQV expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression NEQV expression        %prec LE
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
```

```
| MINUSMARK expression        %prec UMINUS
  {
     $$.xten_type_s = [REF_SYMBOL, "-", "(", $2.xten_type_s, ")"];


      stbl_build1($2);
      passdn_stbl1($2);


      !visibility information
      passdn_2($2, visible_types, visible_names);

      !error_messages
      passup_1($2, error_msgs);
  }
| expression PLUSMARK expression     %prec PLUS
  {
      $$.xten_type_s = [REF_SYMBOL, "+", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression MINUSMARK expression     %prec MINUS
  {
      $$.xten_type_s = [REF_SYMBOL, "-", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression STARMARK expression     %prec MUL
  {
      $$.xten_type_s = [REF_SYMBOL, "*", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression SLASH expression     %prec DIV
  {
      $$.xten_type_s = [REF_SYMBOL, "/", "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];
```

```
        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression MOD expression        %prec MOD
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression EXP expression        %prec EXP
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression U expression        %prec U
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
        passdn2_2($1, $3, visible_types, visible_names);

        !error messages
        $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
    }
  | expression APPEND expression        %prec APPEND
    {
        $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                          ACTUAL_DELIM,$3.xten_type_s, ")" ];

        stbl_build2($1, $3);
        passdn_stbl2($1, $3);

        !visibility information
```

```
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression IN expression        %prec IN
  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| STARMARK expression        %prec STAR
  ! *x is the value of x in the previous state
  {
      $$.xten_type_s = $2.xten_type_s;


      stbl_build1($2);
      passdn_stbl1($2);

      !visibility information
      passdn_2($2, visible_types, visible_names);

      !error_messages
      passup_1($2, error_msgs);
  }
| '$' expression        %prec DOT
  ! $x represents a collection of items rather than just one
  ! s1 = {x, $s2} means s1 = union({x}, s2)
  ! s1 = [x, $s2] means s1 = append([x], s2)
  {
      $$.xten_type_s = $2.xten_type_s;


      stbl_build1($2);
      passdn_stbl1($2);

      !visibility information
      passdn_2($2, visible_types, visible_names);

      !error_messages
      passup_1($2, error_msgs);
  }
| expression RANGE expression        %prec RANGE
  ! x in [a .. b] iff x in {a .. b} iff a <= x <= b
  ! [a .. b] is sorted in increasing order

  {
      $$.xten_type_s = [REF_SYMBOL, $2.%text, "(", $1.xten_type_s,
                        ACTUAL_DELIM,$3.xten_type_s, ")" ];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);
```

```
      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| expression DOTMARK NAME   %prec DOT
  {
     $$.xten_type_s = [REF_SYMBOL, ".", "(", $1.xten_type_s,
                       ACTUAL_DELIM, "\"", $3.%text,"\"", ")"];


      stbl_build1($1);
      passdn_stbl1($1);

      !visibility information
      passdn_2($1, visible_types, visible_names);

      !error_messages
      passup_1($1, error_msgs);
  }
| expression LBRACK expression ']' %prec DOT
  {
      $$.xten_type_s = [REF_SYMBOL, "[", "(", $1.xten_type_s,
                      ACTUAL_DELIM, $3.xten_type_s];

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
  }
| '(' expression ')'
  {
     passup_1($2, xten_type);

      stbl_build1($2);
      passdn_stbl1($2);

      !visibility information
      passdn_2($2, visible_types, visible_names);

      !error_messages
      passup_1($2, error_msgs);
  }
| '(' expression NAME ')'        ! expression with units of measurement
  ! standard time units: NANOSEC MICROSEC MILLISEC SECONDS MINUTES HOURS DAYS
                         WEEKS
  {
      passup_1($2, xten_type);

      stbl_build1($2);
      passdn_stbl1($2);

      !visibility information
      passdn_2($2, visible_types, visible_names);
```

```
      !error_messages
      passup_1($2, error_msgs);
  }
| TIME                ! The current local time, used in temporal events
  {
      $$.xten_type_s = ":real";

      stbl_build0();

      !error messages
      $$.error_msgs_s = "";
  }
| DELAY               ! The time between the triggering event and the response
  {
      $$.xten_type_s = ":real";

      stbl_build0();

      !error messages
      $$.error_msgs_s = "";
  }
| PERIOD        ! The time between successive events of this type
  {
      $$.xten_type_s = ":real";

      stbl_build0();

      !error messages
      $$.error_msgs_s = "";
  }
| literal
  {
      passup_1($1, xten_type);

      stbl_build1($1);
      passdn_stbl1($1);


      !visibility information
      passdn_2($1, visible_types, visible_names);

      !error_messages
      passup_1($1, error_msgs);

  }
| literal '@' actual_name        ! literal with explicit type
  {
      !*** unsure about this one.
      passup_1($1, xten_type);

      stbl_build2($1, $3);
      passdn_stbl2($1, $3);

      !visibility information
      passdn2_2($1, $3, visible_types, visible_names);

      !error messages
      $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];

  }
| '?'               ! An undefined value to be specified later
```

```
          {
              $$.xten_type_s = ":?";

              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        | '!'                ! An undefined and illegal value
          {
             $$.xten_type_s = ":!";

              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        | IF expression THEN expression middle_cases ELSE expression FI
          {
              $$.xten_type_s = $4.xten_type_s;


              stbl_build4($2, $4, $5, $7);
              passdn_stbl4($2, $4, $5, $7);

              !visibility information
              passdn4_2($2, $4, $5, $7, visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$2.error_msgs_s, $4.error_msgs_s,
                                 $5.error_msgs_s, $7.error_msgs_s];
          }
        ;

middle_cases
        : middle_cases ELSE_IF expression THEN expression
          {
              stbl_build3($1, $3, $5);
              passdn_stbl3($1, $3, $5);

              !visibility information
              passdn3_2($1, $3, $5, visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s, $5.error_msgs_s];
          }
        |
          {
              stbl_build0();

              !error messages
              $$.error_msgs_s = "";
          }
        ;

quantifier
        : ALL
          { }
        | SOME
          { }
        | NUMBER
```

```
        { }
      | SUM
        { }
      | PRODUCT
        { }
      | SET
        { }
      | MAXIMUM
        { }
      | MINIMUM
        { }
      | UNION
        { }
      | INTERSECTION
        { }
      ;

restriction
      : SUCH expression
        {
            stbl_build1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
        }
      |
        {
            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
      ;

literal
      : INTEGER_LITERAL
        {
           $$.xten_type_s = ":integer";

            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
      | REAL_LITERAL
        {
            $$.xten_type_s = ":real";

            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
        }
      | CHAR_LITERAL
        {
            $$.xten_type_s = ":char";

            stbl_build0();
```

```
        !error messages
        $$.error_msgs_s = "";
    }
  | STRING_LITERAL
    {
        $$.xten_type_s = ":string";

        stbl_build0();

        !error messages
        $$.error_msgs_s = "";
    }
  | '#' NAME                                  ! enumeration type literal
    {
       $$.xten_type_s = ":enumeration";

        stbl_build0();

        !error messages
        $$.error_msgs_s = "";
    }
  | LBRACK expressions ']'                     ! sequence literal
    {
        $$.xten_type_s = [":sequence{",$2.xten_type_s, "}"];

        stbl_build1($2);
        passdn_stbl1($2);

        !visibility information
        passdn_2($2, visible_types, visible_names);

        !error_messages
        passup_1($2, error_msgs);

    }
  | '{' expressions '}'                          ! set literal
    {
        $$.xten_type_s = [":set{", $2.xten_type_s, "}"];


        stbl_build1($2);
        passdn_stbl1($2);

        !visibility information
        passdn_2($2, visible_types, visible_names);

        !error_messages
        passup_1($2, error_msgs);
    }
  | '{' expressions ';' expression '}'         ! map literal
    {
       $$.xten_type_s = ["map{", $2.xten_type_s, ACTUAL_DELIM,
                         $4.xten_type_s, "}"];


        stbl_build2($2, $4);
        passdn_stbl2($2, $4);

        !visibility information
        passdn2_2($2, $4, visible_types, visible_names);
```

```
            !error messages
            $$.error_msgs_s = [$2.error_msgs_s, $4.error_msgs_s];
          }
        | LBRACK pair_list ']'                    ! tuple literal
          {
             $$.xten_type_s = [ "tuple{", $2.xten_type_s, "}"];

            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
          }
        | '{' pair '}'                            ! one_of literal
          {
            $$.xten_type_s = $2.type_s;


            stbl_build1($2);
            passdn_stbl1($2);

            !visibility information
            passdn_2($2, visible_types, visible_names);

            !error_messages
            passup_1($2, error_msgs);
          }
        ;
          ! relation literals are sets of tuples

expressions
        : expression_list
          {
             passup_1($1, xten_type);

            stbl_build1($1);
            passdn_stbl1($1);

            !visibility information
            passdn_2($1, visible_types, visible_names);

            !error_messages
            passup_1($1, error_msgs);
          }
        |
          {
             $$.xten_type_s = "";


            stbl_build0();

            !error messages
            $$.error_msgs_s = "";
          }
        ;
```

```
pair_list
        : pair_list ',' pair
          {
              $$.xten_type_s = [$1.xten_type_s, PAIR_DELIM, $3.xten_type_s];

              stbl_build2($1, $3);
              passdn_stbl2($1, $3);

              !visibility information
              passdn2_2($1, $3, visible_types, visible_names);

              !error messages
              $$.error_msgs_s = [$1.error_msgs_s, $3.error_msgs_s];
          }
        | NAME pair
          {
              $$.xten_type_s = [$1.%text, "::", $2.type_s, PAIR_DELIM, $2.xten_type_s];
              stbl_build1($2);
              passdn_stbl1($2);

              !visibility information
              passdn_2($2, visible_types, visible_names);

              !error_messages
              passup_1($2, error_msgs);
          }
        | pair
          {
              passup_1($1, xten_type);

              stbl_build1($1);
              passdn_stbl1($1);

              !visibility information
              passdn_2($1, visible_types, visible_names);

              !error_messages
              passup_1($1, error_msgs);
          }
        ;

pair
        : NAME BIND expression
          {
              $$.xten_type_s = [$1.%text, "::", $3.xten_type_s];
              $$.type_s = $3.xten_type_s;

              stbl_build1($3);
              passdn_stbl1($3);

              !visibility information
              passdn_2($3, visible_types, visible_names);

              !error_messages
              passup_1($3, error_msgs);
          }
        ;
```

```
operator_list
        : operator_list operator_symbol
          {
              passdn_3($1, xref_value, message_fargs, ip_mcmxref);
              passdn_stbl1($1);

              $$.d_error_s = check_complex_decl($$.ip_mcmxref_i, $2.operator_text_s,
                             $$.message_fargs_i, $2.line_s, $$.stbl_names_i);
              !modified version of #mk_complex_decl
              $$.ip_mcmxref_s  = ($$.d_error_s == NULL_STRING)
                        -> ($1.ip_mcmxref_s ($2.operator_text_s) == NULL_STRING)
                           -> {($2.operator_text_s : [$$.message_fargs_i,
                                 XREF_DELIMITER, i2s($$.xref_value_i)])} +|
                                 $1.ip_mcmxref_s
                           '#'{($2.operator_text_s : [$1.ip_mcmxref_s
                                                      ($2.operator_text_s),
                                PATTERN_DELIMITER, $$.message_fargs_i,
                                       XREF_DELIMITER, i2s($$.xref_value_i)])} +|
                                       $1.ip_mcmxref_s
                      '#' $1.ip_mcmxref_s;

              !error messages
              $$.error_msgs_s = [ $1.error_msgs_s, $$.d_error_s];

          }
        | operator_symbol
          {
             $$.error_msgs_s = check_complex_decl($$.ip_mcmxref_i, $1.operator_text_s,
                               $$.message_fargs_i, $1.line_s, $$.stbl_names_i);
             mk_complex_decl($$.ip_mcmxref, $$.error_msgs_s, $1.operator_text_s,
                             $$.xref_value_i, $$.message_fargs_i);
          }
        ;

operator_symbol
        : NOT
          {
              $$.operator_text_s = $1.%text;
              $$.line_s = $1.%line;
          }
        | AND
          {
              $$.operator_text_s = $1.%text;
              $$.line_s = $1.%line;
          }
        | OR
          {
              $$.operator_text_s = $1.%text;
              $$.line_s = $1.%line;
          }
        | IMPLIES
          {
              $$.operator_text_s = $1.%text;
              $$.line_s = $1.%line;
          }
        | IFF
          {
              $$.operator_text_s = $1.%text;
              $$.line_s = $1.%line;
          }
```

```
| LT
  {
      $$.operator_text_s = "<";
      $$.line_s = $1.%line;
  }
| GT
  {
      $$.operator_text_s = ">";
      $$.line_s = $1.%line;
  }
| EQUALS
  {
      $$.operator_text_s = "=";
      $$.line_s = $1.%line;
  }
| LE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| GE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NLT
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NGT
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NLE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NGE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| EQV
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| NEQV
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
```

```
| PLUSMARK
  {
      $$.operator_text_s = "+";
      $$.line_s = $1.%line;
  }
| MINUSMARK
  {
      $$.operator_text_s = "-";
      $$.line_s = $1.%line;
  }
| STARMARK
  {
      $$.operator_text_s = "*";
      $$.line_s = $1.%line;
  }
| SLASH
  {
      $$.operator_text_s = "/";
      $$.line_s = $1.%line;
  }
| MOD
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| EXP
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| U
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| APPEND
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| IN
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| RANGE
  {
      $$.operator_text_s = $1.%text;
      $$.line_s = $1.%line;
  }
| DOTMARK
  {
      $$.operator_text_s = ".";
      $$.line_s = $1.%line;
  }
| LBRACK
  {
      $$.operator_text_s = "[";
      $$.line_s = $1.%line;
  }
;
```

APPENDIX C - SYNTACTIC ERROR PRODUCTIONS.

This Appendix contains the syntactic error productions that were developed for version 1.5 of the SPEC grammar. This version was five versions prior to the version used for the type checker, but the methodology used to implement the error productions is fully applicable to the newest version of the grammar. For the type checker to be a fully functional tool, these error productions must be integrated into the final product. In this way, syntactic and semantic errors could be identified concurrently.

```
! version stamp $Header: spec.k,v 1.5 88/02/16 13:27:58 berzins Exp $
! In the grammar, comments go from a "!" to the end of the line.
! Terminal symbols are entirely upper case or enclosed in single quotes (').
! Nonterminal symbols are entirely lower case.
! Lexical character classes start with a captial letter and are enclosed in {}.
! In a regular expression, x+ means one or more x's.
! In a regular expression, x* means zero or more x's.
! In a regular expression, [xyz] means x or y or z.
! In a regular expression, [^xyz] means any character except x or y or z.
! In a regular expression, [a-z] means any character between a and z.
! In a regular expression, . means any character except newline.

! definitions of lexical classes

%define     Digit                                                   :[0-9]
%define     Int                                                     :{Digit}+
%define     Letter                                                  :[a-zA-Z]
%define     Alpha                                                   :({Letter}|{Digit}|"_")
%define     Blank                                                   :[ \t\n]
%define Quote                                                       :["]
%define Backslash                                                   :"\\"
%define     Char        :([^"\\]|{Backslash}{Quote}|{Backslash}{Backslash})

! definitions of white space and comments

                                                                    :{Blank}+
                                                                    :"--".*"\n"

! definitions of compound symbols and keywords

AND                                                                 :"&"
OR                                                                  :"|"
NOT                                                                 :"~"
IMPLIES                                                             :"=>"
IFF                                                                 :"<=>"

LE                                                                  :"<="
GE                                                                  :">="
NE                                                                  :"~="
NLT                                                                 :"~<"
NGT                                                                 :"~>"
```

```
NLE                                    :"~<="
NGE                                    :"~>="
EQV                                    :"=="
NEQV                                   :"~=="

RANGE                                  :".."
APPEND                                 :"||"
MOD                                    :{Backslash}|MOD
EXP                                    :"**"

BIND                                   :"::"
ARROW                                  :"->"

IF                                     :IF
THEN                                   :THEN
ELSE                                   :ELSE
IN                                     :IN
U                                      :U

ALL                                    :ALL
SOME                                   :SOME
NUMBER                                 :NUMBER
SUM                                    :SUM
PRODUCT                                :PRODUCT
SET                                    :SET
MAXIMUM                                :MAXIMUM
MINIMUM                                :MINIMUM
UNION                                  :UNION
INTERSECTION                           :INTERSECTION
SUCH                                   :SUCH{Blank}*THAT
ELSE_IF                                :ELSE{Blank}*IF

AS                                     :AS
CHOOSE                                 :CHOOSE
CONCEPT                                :CONCEPT
DEFINITION                             :DEFINITION
DELAY                                  :DELAY
DO                                     :DO
END                                    :END
EXCEPTION                              :EXCEPTION
EXPORT                                 :EXPORT
FI                                     :FI
FOREACH                                :FOREACH
FROM                                   :FROM
FUNCTION                               :FUNCTION
GENERATE                               :GENERATE
HIDE                                   :HIDE
IMPORT                                 :IMPORT
INHERIT                                :INHERIT
INITIALLY                              :INITIALLY
INSTANCE                               :INSTANCE
INVARIANT                              :INVARIANT
ITERATOR                               :ITERATOR
MACHINE                                :MACHINE
MESSAGE                                :MESSAGE
MODEL                                  :MODEL
OD                                     :OD
OF                                     :OF
OPERATOR                               :OPERATOR
OTHERWISE                              :OTHERWISE
PERIOD                                 :PERIOD
```

```
RENAME                                                  :RENAME
REPLY                                                   :REPLY
SEND                                                    :SEND
STATE                                                   :STATE
TEMPORAL                                                :TEMPORAL
TIME                                                    :TIME
TO                                                      :TO
TRANSACTION                                                 :TRANSACTION
TRANSITION                                              :TRANSITION
TYPE                                                    :TYPE
VALUE                                                   :VALUE
VIRTUAL                                                 :VIRTUAL
WHEN                                                    :WHEN
WHERE                                                   :WHERE

SECONDS                                                 :SECONDS
MINUTES                                                 :MINUTES
HOURS                                                   :HOURS
DAYS                                                    :DAYS
WEEKS                                                   :WEEKS
NANOSEC                                                 :NANOSEC
MICROSEC                                                :MICROSEC
MILLISEC                                                :MILLISEC

INTEGER_LITERAL                                         :{Int}
REAL_LITERAL                                                :{Int}"."{Int}
CHAR_LITERAL                                                :"'"."'"
STRING_LITERAL                                          :{Quote}{Char}*{Quote}

NAME                                                    :{Letter}{Alpha}*

! operator precedences
! %left means 2+3+4 is (2+3)+4.

%left                       ';', IF, DO, EXCEPTION, NAME, SEMI;
%left                       ',', COMMA;
%left                       SUCH;
%left                       IFF;
%left                       IMPLIES;
%left                       OR;
%left                       AND;
%left                       NOT;
%left                       '<', '>', '=', LE, GE, NE, NLT, NGT, NLE, NGE, EQV, NEQV;
%nonassoc                   IN, RANGE;
%left                       U, APPEND;
%left                       '+', '-', PLUS, MINUS;
%left                       '*', '/', MUL, DIV, MOD;
%left                       UMINUS;
%left                       EXP;
%left                       '$', '[', '(', '{', '.', DOT, WHERE;
%left                       STAR;

%%
!attribute declarations

%%
! productions of the grammar

start
            : spec
          {
```

```
          }
            ;

spec
            : spec module
          {
          }
            |
          {
          }
            | spec error module
          {
          }
        ;

              ! A production with nothing after the "|" means the empty string
              ! is a legal replacement for the left hand side.

module
            : function
          {
          }
            | machine
          {
          }
            | type
          {
          }
            | definition
          {
          }
            | instance                                  ! of a generic module
          {
          }
        ;

function
            : optionally_virtual FUNCTION interface messages concepts END
          {
          }
        | optionally_virtual FUNCTION error messages concepts END
            {

            }
        | optionally_virtual FUNCTION interface error
            {
            }
        ;
              ! Virtual modules are for inheritance only, never used directly.

machine
            : optionally_virtual MACHINE interface state messages transactions temporals
concepts END
          {
          }
        | optionally_virtual MACHINE error state messages transactions temporals concepts
END
          {

          }
        | optionally_virtual MACHINE interface error
```

```
          {

          }
        ;

type
            : optionally_virtual TYPE interface model messages transactions temporals
concepts END
          {
          }
        | optionally_virtual TYPE error model messages transactions temporals concepts END
          {

          }
        | optionally_virtual TYPE interface error
          {

          }
        ;

definition
            : DEFINITION interface concepts END
          { }
        | DEFINITION error concepts END
          {
          }
        | DEFINITION interface error
          {
          }
        ;

instance
            : optionally_virtual INSTANCE parametrized_name '=' parametrized_name hide
renames END
          {
          }
        | optionally_virtual INSTANCE error '=' parametrized_name hide renames END
          {

          }
        | optionally_virtual INSTANCE parametrized_name error END
          {


          }
        | optionally_virtual INSTANCE parametrized_name '=' error END
          {

          }
        | optionally_virtual INSTANCE parametrized_name '=' parametrized_name error
          {

          }
        ;
             ! For making instances or partial instantiations of generic modules,
             ! and for making interface adjustments to reusable components
             ! by hiding or changing some names.
```

```
interface
          : NAME formal_parameters inherits imports export
          {
          }
        ;

            ! This part describes the static aspects of a module's interface.
            ! The dynamic aspects of the interface are described in the messages.
            ! A module is generic iff it has parameters.
            ! The parameters can be constrained by a WHERE clause.
            ! A module can inherit the behavior of other modules.
            ! A module can import concepts from other modules.
            ! A module can export concepts for use by other modules.

inherits
          : inherits INHERIT parametrized_name hide renames
          {
          }
          |
          {
          }
        | inherits INHERIT error
          {

          }
        ;

            ! ancestors are generalizations or simplified views of a module
            ! an actor inherits all of the behavior of its ancestors

hide
          : HIDE name_list
          {
          }
          |
            { }
        | HIDE error
          {

          }
        ;

            ! Useful for providing limited views of an actor.
            ! Different user classes may see different views of a system.
            ! Messages and concepts can be hidden.

renames
          : renames RENAME NAME AS NAME
          {
          }
          |
          {
          }
        | renames RENAME error AS NAME
          {

          }
        | renames RENAME NAME error NAME
          {

          }
```

```
        | renames RENAME NAME AS error
          {

          }
        ;

            ! Renaming is useful for preventing name conflicts when inheriting
            ! from multiple sources, and for adapting modules for new uses.
            ! The parameters, model and state components, messages, exceptions,
            ! and concepts of an actor can be renamed.

imports
            : imports IMPORT name_list FROM parametrized_name
          {
          }
            |
          {
          }
        | imports IMPORT error FROM parametrized_name
          {

          }
        | imports IMPORT name_list error parametrized_name
          {

          }
        | imports IMPORT name_list FROM error
          {

          }
        ;

export
            : EXPORT name_list
          {
          }
            |
          {
          }
        | EXPORT error
          {

          }
        ;

messages
            : messages message
          {
          }
            |
          {
          }
            ;

message
            : MESSAGE message_header operator response
          {
          }
        | MESSAGE message_header error
          {
```

```
        }
      ;

response
        : response_body
        {
        }
        | response_cases
        {
        }
        ;

response_cases
        : WHEN expression_list response_body response_cases
        {
        }
        | OTHERWISE response_body
        {
        }
      | WHEN error response_body response_cases
        {
        }
      | WHEN expression_list response_body error
        {

        }
      | OTHERWISE error
        {

        }
      ;

response_body
      : opt_choose opt_reply opt_sends opt_transition
        { }
      ;

opt_choose
        : CHOOSE '(' field_list restriction ')'
        {
        }
      |
        { }
      | CHOOSE error field_list restriction
        {

        }
      | CHOOSE '(' error ')'
        { }
      | CHOOSE '(' field_list restriction error
        {

        }
      ;


opt_reply
        : REPLY message_header where
        {
        }
        | GENERATE message_header where       ! used in iterators
```

```
          {
          }
        |
          { }
        ;

opt_sends
        : sends
          { }
        |
          { }
        ;

sends
            : sends send
          {
          }
            | send
          {
          }
            ;

send
            : SEND message_header TO parametrized_name where foreach
          {
          }
        | SEND message_header error parametrized_name where foreach
          {

          }
        | SEND message_header TO error
          {

          }
        ;

opt_transition
        : transition
          { }
        |
          { }
        ;

transition
            : TRANSITION expression_list                ! for describing state changes
          {
          }
        | TRANSITION error
          {

          }
        ;

message_header
            : optional_exception optional_name formal_arguments
          {
          }
            ;

where
            : WHERE expression_list
```

```
            {
            }
              |       %prec SEMI     ! must have a lower precedence than WHERE
            {
            }
          | WHERE error
            {

            }
        ;

optionally_virtual
            : VIRTUAL
            {
            }
              |
            {
            }
              ;

optional_exception
            : EXCEPTION
            {
            }
              |                                              %prec SEMI
            {

            }
              ;

operator
            : operator OPERATOR operator_list
            {
            }
              |
            {
            }
          | operator OPERATOR error
            {

            }
        ;

foreach
            : FOREACH '(' field_list restriction ')'
            {
            }
              |
            {
            }
          | FOREACH error
            {

            }
          | FOREACH '(' error
            {

            }
        ;
              ! FOREACH is used to describe a set of messages to be sent.
```

```
concepts
            : concepts concept
          {
          }
            |
          {
          }
        ;

concept
            : CONCEPT NAME formal_parameters ':' type_spec where
              ! constants
          {
          }
            | CONCEPT NAME formal_parameters formal_arguments where VALUE formal_arguments
where
              ! functions
          {
          }
        | CONCEPT error formal_parameters formal_arguments where VALUE formal_arguments
where
          {
          }
        | CONCEPT NAME error VALUE formal_arguments where
          {
          }
        | CONCEPT NAME error ':' type_spec where
          {
          }
        | CONCEPT NAME formal_parameters ':' error
          {
          }
        | CONCEPT NAME formal_parameters formal_arguments where VALUE error
          {
          }
        ;

model       ! data types have conceptual models for values
            : MODEL formal_arguments invariant
          {
          }
            | MODEL formal_arguments invariant initially
              ! initially clause specifies automatic variable initialization
          {
          }
        | MODEL error invariant initially
          { }
        | MODEL formal_arguments invariant error
          {
          }
        ;

state       ! machines have conceptual models for states
            : STATE formal_arguments invariant initially
          {
          }
        | STATE error invariant initially
          { }
        | STATE formal_arguments invariant error
          {
          }
```

```
        ;

invariant   ! invariants are true in all states
            : INVARIANT expression_list
          {
          }
        | INVARIANT error
          {
          }
        ;

initially   ! initial conditions are true only at the beginning
            : INITIALLY expression_list
          {
          }
        | INITIALLY error
          {
          }
        ;

transactions
            : transactions transaction
          {
          }
            |
          {
          }
            ;

transaction
            : TRANSACTION parametrized_name '=' action_expression where
          {
          }
        | TRANSACTION error '=' action_expression where
          {
          }
        | TRANSACTION parametrized_name error
          {
          }
        | TRANSACTION parametrized_name '=' error
          {
          }
        ;
              ! Transactions are atomic.
              ! The where clause can specify timing constraints.

action_expression
            : action_expression ';' action_list    %prec SEMI    ! sequence
          {
          }
            | action_list
          {
          }
        | action_expression ';' error
          {
          }
        ;

action_list
            : action_list action_list    %prec STAR    ! parallel
          {
```

```
          }
            | IF alternatives FI     ! choice
          {
          }
            | DO alternatives OD     ! repetition
          {
          }
            | parametrized_name     ! a normal message
          {
          }
            | EXCEPTION parametrized_name     ! an exception message
          {
          }
        | IF error FI
          {
          }
        | DO error OD
          {
          }
        | EXCEPTION error
          {
          }
        | IF alternatives error
          {
          }
        | DO alternatives error
          {
          }
        ;

alternatives
            : alternatives OR guard action_expression
          {
          }
            | guard action_expression
          {
          }
        | alternatives OR error
          {
          }
        ;

guard
            : WHEN expression ARROW
          {
          }
            |
          {
          }
        | WHEN error ARROW
          {
          }
        | WHEN expression error
          {
          }
        ;

temporals
            : temporals temporal
          {
          }
```

```
            |
          {
          }
        ;
temporal
            : TEMPORAL NAME where response
          {
          }
        | TEMPORAL error
          {
          }
        | TEMPORAL NAME error
          {

          }
        ;
              ! Temporal events are trigged at absolute times,
              ! in terms of the local clock of the actor.
              ! The "where" describes the triggering conditions
              ! in terms of "TIME" and "PERIOD".

formal_parameters    ! parameter values are determined at specification time
        : '{' field_list '}' where
          {
          }
            |
          {
          }
        | '{' error '}' where
          {
          }
        | '{' field_list error where
          {
          }
        ;

formal_arguments    ! arguments are evaluated at run-time
            : '(' field_list ')'
          {
          }
            |
          {
          }
        | '(' error ')'
          {
          }
        | '(' field_list error
          {
          }
        ;


field_list
            : field_list ',' field
          {
          }
            | field
          {
          }
        | field_list ',' error
          { }
```

```
        | error ',' field
          {
          }
        ;

field
            : name_list ':' type_spec
          {
          }
            | '$' NAME ':' type_spec
          {
          }
            | '?'
          {
          }
        | name_list ':' error
          {
          }
        | '$' error ':' type_spec
          {
          }
        | '$' NAME error type_spec
          {
          }
        | '$' NAME ':' error
          {
          }
        | name_list error type_spec
          { }
        ;

type_spec
            : parametrized_name                          ! name of a data type
          {
          }
            | TYPE actual_parameters
          {
          }
            | FUNCTION actual_parameters
          {
          }
            | MACHINE actual_parameters
          {
          }
            | ITERATOR actual_parameters
          {
          }
            | '?'
          {
          }
        ;

name_list
            : name_list NAME
          {
          }
            | NAME
          {
          }
            ;
```

```
optional_name
            : NAME formal_parameters
            {
            }
              |
            {
            }
              ;

parametrized_name
            : NAME actual_parameters
            {
            }
              ;

actual_parameters    ! parameter values are determined at specification time
          : '{' arg_list '}'
            {
            }
              |    %prec SEMI    ! must have a lower precedence than '{'
            {
            }
          | '{' error '}'
            {
            }
          | '{' arg_list error
             { }
          ;

actual_arguments    ! arguments are evaluated at run-time
            : '(' arg_list ')'
            {
            }
              |    %prec SEMI    ! must have a lower precedence than '('
            {
            }
          | '(' error ')'
            {
            }
          | '(' arg_list error
            { }
          ;

arg_list
            : arg_list ',' arg                          %prec COMMA
            {
            }
              | arg
            {
            }
          | arg_list ',' error
            {
            }
          ;

arg
            : expression
            {
            }
              | pair
            {
```

```
          }
            ;

expression_list
            : expression_list ',' expression          %prec COMMA
          {
          }
            | expression
          {
          }
        | expression_list ',' error
          {
          }
        | error ',' expression
          { }
        ;

expression
            : quantifier '(' field_list restriction BIND expression ')'
          {
          }
            | parametrized_name actual_arguments
          {
          }
            | parametrized_name '@' parametrized_name actual_arguments
          {
          }
            | NOT expression                          %prec NOT
          {
          }
            | expression AND expression               %prec AND
          {
          }
            | expression OR expression                %prec OR
          {

          }
            | expression IMPLIES expression           %prec IMPLIES
          {

          }
            | expression IFF expression               %prec IFF
          {

          }
            | expression '<' expression               %prec LE
          {

          }
            | expression '>' expression               %prec LE
          {

          }
            | expression '=' expression               %prec LE
          {

          }
            | expression LE expression                %prec LE
          {

          }
```

```
  | expression GE expression                %prec LE
{

}
  | expression NE expression                %prec LE
{

}
  | expression NLT expression               %prec LE
{

}
  | expression NGT expression               %prec LE
{

}
  | expression NLE expression               %prec LE
{

}
  | expression NGE expression               %prec LE
{

}
  | expression EQV expression               %prec LE
{

}
  | expression NEQV expression              %prec LE
{

}
  | '-' expression                          %prec UMINUS
{
}
  | expression '+' expression               %prec PLUS
{

}
  | expression '-' expression               %prec MINUS
{

}
  | expression '*' expression               %prec MUL
{

}
  | expression '/' expression               %prec DIV
{

}
  | expression MOD expression               %prec MOD
{

}
  | expression EXP expression               %prec EXP
{

}
  | expression U expression                 %prec U
{
```

```
}
  | expression APPEND expression                %prec APPEND
{

}
  | expression IN expression                    %prec IN
{

}
  | '*' expression                              %prec STAR
    ! *x is the value of x before a transition
    ! x is the value after the transition
{
}
  | '$' expression                              %prec DOT
    ! $x represents a collection of items rather than just one
! s1 = {x, $s2} means s1 = union({x}, s2)
    ! s1 = [x, $s2] means s1 = append([x], s2)
{
}
  | expression RANGE expression                 %prec RANGE
! x in [a .. b] iff x in {a .. b} iff a <= x <= b
    ! [a .. b] is sorted in increasing order
{

}
  | expression '.' NAME                         %prec DOT
{
}
  | expression '[' expression ']'               %prec DOT
{

}
  | '(' expression ')'
{
}
  | '(' expression units ')'                          ! timing expression
{
}
  | TIME    ! The current local time, used in temporal events
{
}
  | DELAY    ! The time between the triggering event and the response
{

}
  | PERIOD    ! The time between successive events of this type
{

}
  | literal
{
}
  | literal '@' parametrized_name               ! literal with explicit type
{
}
  | '?'       ! An undefined value to be specified later
{

}
```

```
        | '!'       ! An undefined and illegal value
        {

        }
        | IF expression THEN expression middle_cases ELSE expression FI
        {
        }
      | expression '>' error
        { }
      | error '<' expression
        { }
      | quantifier '(' error ')'
        {
        }
      | quantifier error
        {
        }
      | quantifier '(' field_list restriction BIND error ')'
        {
        }
      | parametrized_name '@' error
        {
        }
      | NOT error
        {
        }
      | '-' error
        {
        }
      | '*' error
        {
        }
      | '$' error
        {
        }
      | '(' error ')'
        {
        }
      | IF error THEN expression middle_cases ELSE expression FI
        { }
      | IF expression THEN error middle_cases ELSE expression FI
        { }
      | IF expression THEN expression middle_cases ELSE error
        {
        }
      | literal '@' error
        {
        }
      | '(' expression units error
        {
        }
      ;

middle_cases
        : middle_cases ELSE_IF expression THEN expression
        {
        }
        |
        {
        }
      | middle_cases ELSE_IF error
```

```
            {
            }
          | middle_cases ELSE_IF expression THEN error
            {
            }
          ;

quantifier
              : ALL
            {
            }
              | SOME
            {

            }
              | NUMBER
            {

            }
              | SUM
            {

            }
              | PRODUCT
            {

            }
              | SET
            {

            }
              | MAXIMUM
            {

            }
              | MINIMUM
            {

            }
              | UNION
            {

            }
              | INTERSECTION
            {

            }
              ;

restriction
              : SUCH expression
            {
            }
              |
            {

            }
          | SUCH error
            {
            }
        ;
```

```
literal
            : INTEGER_LITERAL
            {

            }
              | REAL_LITERAL
            {

            }
              | CHAR_LITERAL
            {

            }
              | STRING_LITERAL
            {

            }
              | '#' NAME                                    ! enumeration type literal
            {

            }
              | '[' expressions ']'                                ! sequence literal
            {
            }
            | '{' expressions '}'                  ! set literal
            {
            }
            | '{' expression ';' expressions '}'   ! map literal
            {
            }
              | '[' pair_list ']'                                  ! tuple literal
            {
            }
            | '{' pair '}'                         ! one_of literal
            {
            }
            | '#' error
            {
            }
            | '[' error ']'
            {
            }
            | '{' error '}'
            {
            }
            | '{' expression ';' error '}'
            {
            }
            | '[' pair_list error
            { }
            | '[' expression_list error
            { }
            ;

                  ! relation literals are sets of tuples

expressions
            : expression_list
            {
            }
```

```
            |
          {

          }
            ;

pair_list
            : pair_list ',' pair
          {
          }
            | NAME pair
          {
          }
            | pair
          {
          }
        | pair_list ',' error
          {
          }
        ;

pair
            : NAME BIND expression
          {
          }
        | NAME BIND error
          {
           }
        ;

units
            : NANOSEC
          {

          }
            | MICROSEC
          {

          }
            | MILLISEC
          {

          }
            | SECONDS
          {

          }
            | MINUTES
          {

          }
            | HOURS
          {

          }
            | DAYS
          {

          }
            | WEEKS
          {
```

```
          }
            ;

operator_list
            : operator_list operator_symbol
          {
          }
            | operator_symbol
          {
          }
        | operator_list error operator_symbol
          {
          }
        ;

operator_symbol
            : NOT
          {

          }
            | AND
          {

          }
            | OR
          {

          }
            | IMPLIES
          {

          }
            | IFF
          {

          }
            | '<'
          {

          }
            | '>'
          {

          }
            | '='
          {

          }
            | LE
          {

          }
            | GE
          {

          }
            | NE
          {

          }
```

```
    | NLT
  {

  }
    | NGT
  {

  }
    | NLE
  {

  }
    | NGE
  {

  }
    | EQV
  {

  }
    | NEQV
  {

  }
    | '+'
  {

  }
    | '-'
  {

  }
    | '*'
  {

  }
    | '/'
  {

  }
    | MOD
  {

  }
    | EXP
  {

  }
    | U
  {

  }
    | APPEND
  {

  }
    | IN
  {

  }
    | RANGE
```

```
{

}
  | '.'
{

}
  | '['
{

 }
  ;
```

APPENDIX D - TYPE CHECKER ATTRIBUTES.

This Appendix contains a list of all of the attribute descriptive names used in the implementation of the type checker with their purpose. This list is not all encompassing. Various attributes that are slight variations of the below named attributes have been left out. All of the attributes not included have an "ip" prefix with a "main" name corresponding to a name listed below.

Attribute Name	Attribute Purpose
%line	A Kodiyak predefined attribute yielding the line number in the source text of a terminal symbol.
%text	A Kodiyak predefined attribute yielding the actual text of a terminal symbol.
actual_name_text	The actual text of an actual name.
actual_params	The actual parameters associated with an actual name.
actual_text_s	The actual text associated with a non-terminal.
arg_text	The text of an arg non-terminal.
arglist_text	The text of an argument list (arg_list) non-terminal.
args_i	The arguments associated with a particular non-terminal. This attribute is commonly used in the formal name non-terminal to obtain the arguments associated with the name so a "pattern" may be created.
d_error_s	An attribute containing an error message relating to the declaration of a new name (if any such error exists).
env_i	The current environment of a non-terminal.
error_msgs	This attribute contains all error messages identified in the current and children non-terminals.

fieldpattern	The string containing the non-terminals name and it's type, separated by a predefined delimiter.
global_type	A map containing all global type names and their translation.
ip_lclzd_mcmxref	A localized attribute containing the same information as the mcmxref attribute. Used in the final stages of layer 2 to integrate ip_mcmxref attributes from different modules.
ip_mcmxref	A map containing all module, message and concept names. This map is used in layer 2 to build the layer 3 attribute "stbl". The range of the map contains patterns.
ip_mxref	A map which is being constructed to contain all module names. The range of the map contains the module names pattern and cross reference. This map is used in layer 1 to obtain information needed in the layer 2 table "ip_mcmxref".
local_types_s	The type names that are local to a non-terminal.
line	The line number associated with a non-terminal.
message_name_s	The text of a message's name.
message_fargs_s	The text of the formal arguments of a message.
mod_types	A localized map containing the type names and their translations that are visible. This table and the "global_type" attributes are used to build the "visible_types" map.
module_name	The name of the current module.
mxref_value	The cross reference value of the current module.
name_fargs	The formal arguments associated with a name.
name_text	The text associated with a formal or actual name.
name_type_text	The text of a name's type.
name_type_value	The translated value (obtained from the visible types table) of a name's type.

name_params	The parameters associated with a formal or actual name.
operator_text	The text associated with an operator non-terminal.
signature	The signature (name and arguments) of a non-terminal.
stbl_class	Classes of all names used (e.g. Function, Message, etc.) Accessed by the cross reference value of the name.
stbl_names	All names used throughout the program. Accessed by a cross reference value.
stbl_params	The formal parameters of a name (if any). Accessed by a cross reference value.
stbl_result	The resultant type of a name. Contains extended type information. Accessed by the cross reference value of the name.
text	The text of a non-terminal.
type_name_text	The text of a type_spec's name.
type_name_value	The translated value of a type_spec.
type_table_i	A map coalescing the contents of the "mod_types" and "global_type" tables.
visible_names	A map containing all names visible.
visible_types	A map containing all type names visible.
xref_value	The cross reference value of the current name.
xten_type_	The extended type of a non-terminal.

REFERENCES.

1. Berzins, V., and Luqi, Draft of *Software Engineering with Abstractions: An Integrated Approach to Software Development with Ada*. Addison-Wesley, 1988.

2. Weigand, J. *Design and Implementation of a Pretty Printer for the Functional Specification Language SPEC*, M.S. Thesis, Naval Postgraduate School, June, 1988.

3. Berzins, V., and Gray, M., "Analysis and Design in MSG.84: Formalizing Functional Specifications," *IEEE Transactions in Software Engineering*, v. 11, pp. 657-670, August 1985.

4. Johnson, S., *YACC--Yet Another Compiler Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.

5. Farrow, R. "Generating a Production Compiler for an Attribute Grammar," *IEEE Software*, v. 1, pp. 77-93, October 1984.

6. Herndon, R. *Attribute Grammar Systems for Prototyping Translators and Languages*. Ph. D. Dissertation, University of Minnesota, 1988.

7. Nicholas, C. *Assusring Accessibility of Complex Software Systems*. Ph. D. Dissertation, University of Ohio State, 1988.

8. Aho, A. and Ullman, J., *Principles of Compiler Design*, p. 58, Addison-Wesley, 1979.

9. Knuth, D. "On the Translation of Languages from Left to Right," *Information and Control*, v. 8, pp. 607-639, 1965.

10. Brooks, F. "The Mythical Man-Month," *Datamation*, v. 20, No. 12, pp. 44-52, December 1974.

11. Verner, J and Tate, G., "Estimating Size and Effort in Fourth-Generation Development", *IEEE Software*, pp. 15-22, July 1988.

12. Penello, T. "Very Fast LR Parsing," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pp. 224-233, 1982.

13. University of Helsinki Technical Report A-1978-2, *The Compiler Writing System HLP*, by K. Raiha, M. Saarinen, E. Soisalon-Soininen, and M. Tienari, March 1978.

14. Lorho, B. "Semantic Attribute Processing in the System Delta," *Lecture Notes in Computer Science*, v. 47, pp. 21-40, Springer-Verlag, 1977.

15. Ganzinger, H., Ripken, K., and Wilhelm, R., "Automatic Generation of Multipass Compilers," *Information Processing 77, Proceedings of IFIP Congress 77*, pp. 535-540, North-Holland Publishing Co., 1977.

16. Milton, D., Kirchhoff, L., and Rowland, B., "An ALL(1) Compiler Generator," *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices*, v. 14, No. 8, pp. 152-157, August 1979.

17. Kastens, U., Hutt, B., and Zimmerman, E., "GAG: A Practical Compiler Generator", *Lecture Notes in Computer Science*, v. 141, Springer-Verlag, 1982.

18. Bell Laboratories Computer Science Technical Report 39, *Lex - A Lexical Analyzer Generator*, by M. Lesk and E. Schmidt, October 1975.

19. Naval Postgraduate School Technical Report NPS52-89-029, *A Student's Guide to SPEC*, by V. Berzins and R. Kopas, May 1989.

20. Naval Postgraduate School Technical Report NPS52-87-032, *The Semantics of Inheritance in Spec*, by V. Berzins and Luqi, July 1987.

21. Beebe, D. *The Design and Implementation of a Syntax Directed Editor for the Specification Language Spec*, M.S. Thesis, Naval Postgraduate School, June, 1989.

BIBLIOGRAPHY.

Adorni, G., Boccolatte, A., and Di Manzo, M., "Top-Down Semantic Analysis", *Computer Journal*, v. 27, August 1984.

Berzins, V., Gray, M., and Naumann, D., "Abstraction Based Software Development", *Communications of the ACM*, v. 29 No. 5, pp. 402-415, May 1986.

Booch, G., *Software Engineering with Ada*, 2nd edition, Benjamin/Cummings, 1987.

Cleaveland, Craig. "Building Application Generators," *IEEE Software*, v. 14 pp. 25-33, July 1988.

Farrow, R., "Generating a Production Compiler for an Attribute Grammar," *IEEE Software*, v. 1, October 1984, pp. 77-93.

Fisher, A., *CASE, Using Software Development Tools*, John Wiley & Sons, Inc., 1988.

Herndon, R. and Berzins, V., "The Realizable Benefits of a Language Prototyping Language", *IEEE Transaction on Software Engineering*, v. 14, June 1988, pp. 803-809.

MacLennan, B., *Principles of Programming Languages: Design, Evaluation, and Implementation*, 2nd edition, Holt, Rinehart & Winston, 1987.

Naval Postgraduate School Technical Report NPS52-87-033, *Specifying Large Software Systems in Spec*, by V. Berzins and Luqi, July 1987.

Naval Postgraduate School Technical Report NPS52-88-007, *Generating a Language Translator based on an Attribute Grammar Tool*, by LuQi and R. Herndon, April 1988.

Naval Postgraduate School Technical Report NPS52-88-038, *Languages for Specification, Design and Prototyping*, by V. Berzins and LuQi, September 1988.

Reps, Charles. *Generating Language-Based Environments*. Ph. D. Dissertation, University of Massachusetts, Amherst, 1983.

University of Minnesota Computer Science Department Report Technical Report 85-25, *AG: A Useful Attribute Grammar Translator Generator*, by R. Herndon and V. Berzins, July 1985.

University of Minnesota Computer Science Technical Report 85-37, *The Incomplete AG User's Guide and Reference Manual*, by R. Herndon, October 1985.

INITIAL DISTRIBUTION LIST.

1.	Ada Joint Program Office OUSDRE (R&AT) The Pentagon Washington, D.C. 20301	1
2.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, D.C. 20375-5000	1
3.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) Attn. Dr. Jacob Schwartz 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
4.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) Attn. Dr. Squires 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
5.	Defense Advanced Research Projects Agency (DARPA) Director, Naval Technology Office 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
6.	Defense Advanced Research Projects Agency (DARPA) Director, Tactical Technology Office 1400 Wilson Boulevard Arlington, Virginia 22209-2308	1
7.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
8.	Dr. Amiram Yehudai Tel Aviv University School of Mathematical Sciences Department of Computer Science Tel Aviv, Israel 69978	1

9. Dudley Knox Library — 2
Code 0142
Naval Postgraduate School
Monterey, California 93943-5002

10. Fleet Combat Direction Systems Support Agency — 1
Attn. Mike Reiley, Code 00T
San Diego, California 92147-5081

11. Fleet Combat Direction Systems Support Agency — 1
Attn. George Roberson, Code 8D
San Diego, California 92147-5081

12. International Software Systems Inc. — 1
12710 Research Boulevard, Suite 301
Attn. Dr. R. T. Yeh
Austin, Texas 78759

13. Kestrel Institute — 1
Attn. Dr. C. Green
1801 Page Mill Road
Palo Alto, California 94304

14. Lt. Robert Kopas — 1
Department Head School Class 110
Surface Warfare Officers School Command
Newport, Rhode Island 02841-5012

15. Massachusetts Institute of Technology — 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. B. Liskov
Cambridge, Massachusetts 02139

16. Massachusetts Institute of Technology — 1
Department of Electrical Engineering and Computer Science
545 Tech Square
Attn. Dr. J. Guttag
Cambridge, Massachusetts 02139

17. MCC AI Laboratory — 1
Attn. Dr. Michael Gray
3500 West Balcones Center Drive
Austin, Texas 78759

18. Office of Naval Research — 1
Computer Science Division, Code 1133
Attn. Dr. R. Wachter
800 N. Quincy Street
Arlington, Virginia 22217-5000

19. Office of the Secretary of Defense — 1
R & AT/S & CT, RM 3E114
STARS Program Office
Washington, D.C. 20301

20. Oregon Graduate Center — 1
Portland (Beaverton)
Attn. Dr. R. Kieburtz
Portland, Oregon 97005

21. Software Group, MCC — 1
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759

22. University of Pittsburgh — 1
Department of Computer Science
Attn. Dr. Alfs Berztiss
Pittsburgh, Pennsylvania 15260

23. Purdue University — 1
Department of Computer Science
West Lafayette, Indiana 47906

24. The Ohio State University — 1
Department of Computer and Information Science
Attn. Dr. Charles Nicholas
2036 Neil Ave Mall
Columbus, Ohio 43210-1277

25. Berzins — 4
Code 52Bz
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

Thesis
K7922 Kopas
c.1 The design and implementation of a specification language type checker.

thesK7822
The design and implementation of a speci
3 2768 000 82907 1
DUDLEY KNOX LIBRARY